Porting TinyOS-2.x to the Chipcon CC2430
Individual Project (Computing) - UFEEJ4-40-3

Nicholas Beck
1. Abstract

This dissertation outlines the issues involved and difficulties encountered porting TinyOS-2.x, the *de facto* operating system for wireless sensor networks, to the Chipcon CC2430 System-on-Chip platform. Specifically, this included an architecture port of the 8051 processor-core, the CC2430 platform, the RF04EB sensorboard, and custom MARCO sensorboard. Furthermore, the CC2430 requires the use of a toolchain which is not supported by TinyOS.

The CC2430 is composed of an enhanced 8051-core microcontroller and a Chipcon CC2420 2.4GHz RF transceiver. It is one of the first, of few, second generation integrated devices and represents what appears to be an ideal platform for TinyOS.

Core issues that limit the portability of TinyOS-2.x are identified, and suggestions made as to how these might be addressed. Additionally, constraints imposed by the hardware implementation, and the implications imposed by these, are discussed.

The lack of support for second generation platforms in TinyOS has played a large part in research groups not adopting their use. The contributions from this *real* research effort will benefit the Wireless Sensor Network community.

The author has contributed a paper, *Shaping TinyOS to Deal with Evolving Device Architectures: Experiences Porting TinyOS-2.0 to the Chipcon CC2430*, for EMNETS’07 which will be published by the ACM.

The author is a member of the TinyOS 8051 Working Group and has made the source code from this work publicly available in the TinyOS source repository, released under the TinyOS license, at [http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/nixtems](http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/nixtems).

With the exception of radio receive support, this work represents a fully useable port of the CC2430 platform.
## 2. Contents

1. Abstract .........................................................................................................................1
2. Contents.........................................................................................................................2
3. List of Figures ...............................................................................................................6
4. List of Tables.................................................................................................................7
5. List of Code Samples ....................................................................................................8
6. Overview .....................................................................................................................11
   6.1 Project Overview........................................................................................................11
   6.2 Wireless Sensor Networks.....................................................................................11
7. Network Topologies & Medium Access Control.........................................................13
8. Wireless Standards ....................................................................................................14
   9.1 Comparison of Available Mote Platforms...............................................................16
   9.2 Classes of Mote.........................................................................................................17
   9.3 Types of Mote ..........................................................................................................18
      9.3.1 Common Off-The-Shelf (COTS) Motes .............................................................18
         9.3.1.1 Unintegrated.................................................................................................18
         9.3.1.2 System-on-Chip..........................................................................................18
      9.3.2 Custom Hardware Motes .................................................................................19
   9.4 The Chipcon CC2430 Mote ...................................................................................19
      9.4.1 Processor Core ................................................................................................21
      9.4.2 Radio Transceiver ...........................................................................................22
   9.5 Evaluation of the CC2430 as a Mote Platform........................................................22
10. Operating Systems for Wireless Sensor Networks ..................................................22
    10.2 TinyOS ..................................................................................................................24
       10.2.1 NesC ...............................................................................................................25
          10.2.1.1 Wiring ......................................................................................................25
          10.2.1.2 Attributes ..............................................................................................26
          10.2.1.3 Compile-time Constant Functions .........................................................26
       10.2.2 Static Analysis ...............................................................................................26
       10.2.3 Components & Interfaces ..............................................................................27
          10.2.3.1 Interfaces .................................................................................................27
11.10 Timers.........................................................................................................................55
  11.10.1 Hardware Issues........................................................................................................56
  11.10.2 Timer Interfaces........................................................................................................56
  11.10.3 Timer Transformations..............................................................................................57
  11.10.4 Sleep Timer................................................................................................................58
     11.10.4.1 Millisecond Timer............................................................................................60
     11.10.4.2 Implications.....................................................................................................66
     11.10.4.3 Testing the Virtualized Millisecond Timer & Underlying Sleep Timer ..........66
  11.10.5 Timer1 & BusyWaitMicroC......................................................................................66
     11.10.5.1 Testing BusyWaitMicroC & Timer1..............................................................68
  11.10.6 Timer3 & Timer4......................................................................................................68
11.11 SMBus / I²C..................................................................................................................70
11.12 Analogue-to-Digital Converter ADC............................................................................72
  11.12.1 Testing the ADC Subsystem....................................................................................75
  11.12.2 Integrated Temperature Sensor................................................................................75
11.13 UART..............................................................................................................................75
  11.13.1 Printf()..................................................................................................................79
  11.13.2 Devices Connected to Motes...................................................................................79
  11.13.3 Testing the UART Subsystem................................................................................79
11.14 Radio.............................................................................................................................79
12. Future Work......................................................................................................................86
  12.1 Fully Automated Integration of the IAR Compiler............................................................86
  12.2 Further Energy Efficiency Optimisations........................................................................86
     12.2.1 MCU Clock Speed..............................................................................................86
     12.2.2 Direct Memory Addressing (DMA).........................................................................86
  12.3 GPIO & GPIO Interrupts..................................................................................................87
  12.4 GPIO Interrupts.............................................................................................................87
  12.5 Radio Receive Support....................................................................................................87
  12.6 Heterogeneous Mote Communications..........................................................................87
  12.7 Energy Efficiency Comparison with Other Mote Platforms............................................87
13. Evaluation & Conclusion..................................................................................................87
  13.1 Appraisal of the CC2430...............................................................................................87
  13.2 Contributions................................................................................................................88
  13.3 Were the Project Aims Met?..........................................................................................88
  13.4 Personal Reflection on the Project................................................................................89
14. Glossary................................................................................................................................90
Appendix A - Discussion of 8051 Compiler Options..............................................................91
3. List of Figures

Figure 1: Orthodox Lifecycle of WSN Applications ................................................................. 12
Figure 2: Mesh Network Topology - Partially Connected ..................................................... 13
Figure 3: Network Topologies Employed by WSNs [ ] .......................................................... 14
Figure 4: Relationship of ZigBee to 802.15.4 [ ] ................................................................. 15
Figure 5: Classes of Mote Platforms [22] ............................................................................. 17
Figure 6: A Selection of COTS Motes: (left to right) BT Node, Mica2, Tmote Sky, Sun SPOT ... 18
Figure 7: CC2430 SoC Evaluation Module ........................................................................... 19
Figure 8: The SPEC Mote pictured next to a ballpoint pen .................................................. 19
Figure 9: CC2430 Block Diagram [28] .................................................................................. 20
Figure 10: Diagram Legend .................................................................................................... 32
Figure 11: CC2430 Memory Map [28] ................................................................................... 33
Figure 12: Standard TinyOS Build Process ......................................................................... 35
Figure 13: Modified TinyOS Build Process Implemented by WISENET [103] ....................... 38
Figure 14: Modified TinyOS Build Process Implementation for the CC2430 ....................... 38
Figure 15: Original GPIO Subsystem Design ....................................................................... 46
Figure 16: Revised GPIO Subsystem Design ....................................................................... 47
Figure 17: Idealised Design for GPIO on the Chipcon C2430 .............................................. 47
Figure 18: Design of the LEDs Abstraction including Platform-independent Library components .... 49
Figure 19: Single Configuration for GPIO & GPIO Interrupts .............................................. 50
Figure 20: GPIO Interrupt Design ....................................................................................... 51
Figure 21: HPL Control Interface & Components for GPIO & GPIO Interrupts ..................... 52
Figure 22: Interrupt Enable Capabilities of the CC2430 GPIO Port ..................................... 53
Figure 23: GPIO Interrupt Flags on the CC2430 ................................................................. 53
Figure 24: Schematic for the Push Button on the RF04EB Sensorboard [121] ..................... 55
Figure 25: Common Interfaces Used By All Timers ............................................................. 56
Figure 26: Millisecond Timer Design – Lower .................................................................... 59
Figure 27: CC2430 Sleep Timer Registers [28] .................................................................. 61
Figure 28: Service Instance Design Pattern [60] ................................................................. 63
Figure 29: Millisecond Timer Design – Upper .................................................................... 64
Figure 30: Design of BusyWaitMicroC, Realised Through Timer1 ....................................... 67
Figure 31: Timer3 Design; Presents Alarm & Counter Interfaces with Native Size & Precision ... 68
Figure 32: The Dispatcher Design Pattern ........................................................................... 69
Figure 33: Timer3 Transformations to provide 32-bit, 32kHz Alarm & Counter Interfaces .... 70
Figure 34: Data Transfers on SMBus [94] ........................................................................... 70
Figure 35: I2C Subsystem Design ....................................................................................... 71
Figure 36: ADC Configuration Callback Mechanism ......................................................... 73
Figure 37: Design of the ADC Subsystem .......................................................................... 74
Figure 38: The Full UART Subsystem ............................................................................... 75
Figure 39: HPL, HAL & HIL Level UART Interfaces ................................................................. 76
Figure 40: UART Transmission State Machine ................................................................. 77
Figure 41: Chip- & Platform-specific UART Subsystem Design ................................................. 78
Figure 42: Modifications Necessary to Tackle Data Transmission Issue ........................................ 79
Figure 43: Connections Between a Standalone CC2420 Radio & Separate MCU [ ] ......................... 80
Figure 44: Modifications to CC2420 Implementation to decouple it from the SPI Bus – Only configuration components are shown ........................................................................ 81
Figure 45: Interfaces and Generic Components Used by the Chip-specific Components of the Radio Abstraction .......................................................................................... 82
Figure 46: Design of Chip-specific Region of the Radio Abstraction .............................................. 83
Figure 47: RF04EB Evaluation/Sensor Board [121] ................................................................. 100
Figure 48: Push Button Abstraction ....................................................................................... 101
Figure 49: De-bounce Logic .................................................................................................. 102
Figure 50: Design of the LCD Abstraction ................................................................................. 102
Figure 51: Configuration Required for a System with Multiple SMBus Devices ......................... 103
Figure 52: Design of the Potentiometer Abstraction ................................................................. 104
Figure 53: ADC Configuration Callback Mechanism ............................................................. 105
Figure 54: Schematic of the Joystick: Wired Through a Potential Divider to Reduce Pin Count [121] ....................................................................................................................... 105
Figure 55: Design of the Joystick Abstraction ......................................................................... 106
Figure 56: MARCO Sensorboard: Shown Connected to MARCO Robot ..................................... 108
Figure 57: Graphical Representation of Pulse-Width-Modulation [ ] .......................................... 109
Figure 58: Timer3 Compare Register for Channel 0, showing Compare Output Action ............... 111
Figure 59: Peripheral I/O Pin Mapping Options on the CC2430 .................................................. 111
Figure 60: The Placeholder Design Pattern [55] ...................................................................... 112
Figure 61: MARCO Motor Design Incorporating a Basic & Advanced Interface ......................... 113
Figure 62: Design of the Motor Position Abstraction & Interface Definition .............................. 114
Figure 63: MARCO Wheel Motor Position Encoding Pattern .................................................... 115
Figure 64: Bumper Abstraction Design & Interface Definition .................................................. 115
Figure 65: Alternative Abstraction which Passes Responsibility of Wiring to the Client ............... 116
Figure 66: Bumper Abstraction Design & Interface Definition .................................................. 116

4. List of Tables

Table 1: Comparison of WSN Applications & their Implications on the Hardware Platform ........ 13
Table 2: Comparison of Wireless Standards [ ] [10] [13] [14] ................................................ 15
Table 3: Comparison of Available WSN Hardware .................................................................. 16
Table 4: Typical Characteristics & Properties of Mote Classes [22] ........................................ 18
Table 5: CC2430 Current Consumption in Various Power Modes ......................................... 21
Table 6: Wake-up Times from Various Power Modes ............................................................. 21
Table 7: Comparison of Radio Chips Suitable for WSN Platforms [ ] ....................................... 22
Table 8: Comparison of Embedded Operating Systems .................................................. 23
Table 9: Comparison of WSN Operating Systems ....................................................... 23
Table 10: Effect of inlining on application footprint and performance. All sizes are in bytes. The CPU reduction column shows the reduction in CPU cycles spent executing tasks with optimizations enabled [53]. ................................................................................................. 26
Table 11: Optimization effects on timer event handling. This figure shows the breakdown, in CPU cycles, for both work and boundary crossing for timer event handling, which requires 7 module crossings. Optimization reduces the overall cycle count by 38% [53]................................... 27
Table 12: TinyOS File Naming Convention .................................................................... 30
Table 13: Comparison of Register Declarations for Different 8051 Compilers ............... 33
Table 14: Previous TinyOS Port Attempts to Platforms Without GCC Support ................ 34
Table 15: Actions of nesC Compiler Support Scripts................................................... 36
Table 16: SFR & XDATA Memory Access Examples .................................................... 36
Table 17: SFR Declaration ......................................................................................... 36
Table 18: ISR Declaration ......................................................................................... 37
Table 19: ISR Declaration Format Using IAR-specific language Extensions .................. 37
Table 20: Comparison of Necessary Actions by the Syntax Translation Script for the 8051 & Keil/IAR/SDCC Compiler ................................................... 39
Table 21: Additional Actions Required by the Syntax Translation Script ......................... 40
Table 22: CC2430 Clock Facilities ...................................................................... 56
Table 23: A Comparison of ADC Properties on Different MCUs组 ................................................................................................................................. 73
Table 24: ADC Conversion Times ........................................................................ 75
Table 25: Comparison of Integrated and Standalone Versions of the CC2420 Radio Transceiver .......................................................... 80
Table 26: Shorthand Syntax the IAR Compiler Cannot Interpret & the Modifications Required ............ 85
Table 27: Types of Files in the TinyOS Build System: Their File Extensions & Roles ................. 93
Table 28: TinyOS Application Debugging Techniques................................................... 99

5. List of Code Samples

Code Sample 1: Leds Interface File ........................................................................ 27
Code Sample 2: BlinkAppC Configuration File...................................................... 28
Code Sample 3: BlinkC Module File..................................................................... 28
Code Sample 4: Build Script used by T8051WG [102] ............................................. 38
Code Sample 5: Additional specs File, which Overrides Default Behaviour .......... 39
Code Sample 6: Modifications to the Standard Architecture Rules for the 8051 Processor ................................................... 39
Code Sample 7: Compiler Choice through Command-line Option – iar.extra ................ 39
Code Sample 8: The TinyOS-2.x Boot Sequence Implementation – RealMainP .......... 41
Code Sample 9: CC2430 Platform-specific Initialisation - PlatformC ...................... 41
Code Sample 10: The Dictated Interface Signature of the McuSleepC Component ....... 42
Code Sample 11: Power Management Functionality - McuSleepC ......................... 43
Code Sample 12: HIL Layer GeneralIO Interface .................................................. 44
Code Sample 13: Output of Generic Components Generated by the nesC Compiler: from the ButtonToLed Application ..............................................................44
Code Sample 14: Example of Wiring to the GPIO Subsystem from Higher Level Components: PlatfromLedC ........................................................45
Code Sample 15: Atmega128 GPIO Generic Component Instantiation [] ..................................................48
Code Sample 16: Atmega128 GPIO Generic Component Hack [] ..................................................48
Code Sample 17: GpioInterrupt HIL Interface ..................................................................................50
Code Sample 18: Top-level Configuration of the GPIO Interrupt Subsystem ........................................51
Code Sample 19: Parameterised Timer Interfaces .............................................................................56
Code Sample 20: Example Timer Precision Definitions .....................................................................56
Code Sample 21: TransformCounterC's Component Interface Signature ...........................................57
Code Sample 22: Example Usage of TransformCounterC .................................................................57
Code Sample 23: TransformAlarmC's Component Interface Signature .................................................57
Code Sample 24: Example Usage of TransformAlarmC ......................................................................58
Code Sample 25: Suggested naming Convention of Platform-specific Counter & Alarm Components ........................................................58
Code Sample 26: Timer-width Manipulation Macros .........................................................................61
Code Sample 27: Component Member Function InternalOverflow() – Hpl8051SleepTimerP ..............61
Code Sample 28: Timer.get() Operation, Detects Timer Overflow- Hpl8051SleepTimerP ......................62
Code Sample 29: Catch Timer Overflow Events, Trap Method 1 - Hpl8051SleepTimerP ......................62
Code Sample 30: Catch Timer Overflow Events, Trap Method 2 - Hpl8051SleepTimerP ......................63
Code Sample 31: TimerMilliC - Top-level Configuration that Creates a Virtual Millisecond Timer .......65
Code Sample 32: TimerMilliP – Private Configuration that Wires the Single Timer Interface Requested by TimerMilliC to the Parameterised Timer Interface Provided by HilTimerMilliC ........................................65
Code Sample 33: HilTimerMilliC - HIL Component that Provides the Virtualised Millisecond Timer 65
Code Sample 34: Component Interface Signature of BusyWaitMicroC ..............................................66
Code Sample 35: BusyWait Interface Implementation ........................................................................68
Code Sample 36: Suggested naming Convention of Platform-specific Counter & Alarm Components 68
Code Sample 37: Creation of an Alarm Instance .............................................................................69
Code Sample 38: Wiring of Parameterised Compare Interface to Explicit Comparators .......................69
Code Sample 39: SoftwareSmbusC Configuration ............................................................................71
Code Sample 40: LCD Using the Bit-banged SMBus Implementation – LcdC Configuration ..........71
Code Sample 41: ADC HIL Interfaces ...............................................................................................73
Code Sample 42: UART Data Transmission State Machine - i8051UsartP.nc .....................................77
Code Sample 43: Connection-agnostic Interface - Suitable for Integrated and Standalone versions of the CC2430 ..............................................................................81
Code Sample 44: Sample of Radio RAM Implementation ...................................................................84
Code Sample 45: Radio Register Access - Generic Module ..................................................................85
Code Sample 46: Event Signalling through Task Context, because no Interrupt is Generated by Hardware .................................................................................85
Code Sample 47: Exporting Path to Windows Program for Invocation from Cygwin ......................93
Code Sample 48: TinyOS Target File to Include the CC2430 Platform into the Build System - cc2430.target .................................................................................................94
Code Sample 49: Extracts from the 8051 Processor Compilation Rules - mcs51.rules.........................94
Code Sample 50: IAR Specific Options - iar.extra...........................................................................94
Code Sample 51: .platform File for the CC2430 .............................................................................95
Code Sample 52: Additional Rules for use of TinyOS with the IAR Compiler - iar.extra .........................95
Code Sample 53: Selection & Inclusion of Sensorboard........................................................................96
Code Sample 54: Use of .extra File to Select Required Sensorboard – rf04eb.extra .........................96
Code Sample 55: Example TinyOS Application Makefile .....................................................................96
Code Sample 56: A Section of a TinyOS Application – nesC in black, generated C code in red...........98
Code Sample 57: Example of Variable Name-mangling, Generic Component-mangling and Parameterised Interfaces..................................................................................99
Code Sample 58: Placeholder Configuration for the MARCO Motor Abstraction: MotorsC .............112
Code Sample 59: Example Usage of the MotorsC Abstraction..........................................................112
6. Overview

6.1 Project Overview

This document describes the work performed in porting TinyOS, the de facto operating system for wireless sensor networks, to the System-on-Chip (SoC) Chipcon CC2430. This work is real research, whose contributions have benefited the Wireless Sensor Network (WSN) community.

At the time of writing TinyOS-2.x has twelve supported platforms (including variants) which all use one of only three supported microcontrollers. None of the TinyOS platforms utilise integrated SoC devices. By porting an integrated platform an insight into the gains provided by second-generation hardware is possible. Additionally, due to the substantial reduction in size the CC2430 offers, in comparison to current platforms, the ability to create WSN applications that demand a small footprint can be realised.

On undertaking this project it was not known whether it was possible to port TinyOS-2.x to the CC2430. Work performed by other groups prior to this suggested that this may be feasible, however these ports were to TinyOS-1.x, which is substantially different. Therefore the project’s aims consisted of two paths: ascertaining the suitability of the chip as a TinyOS-2.x platform, and an idealised set of development aims.

The idealised aims involved:

- Modifying TinyOS to handle an unsupported toolchain, so that executables could be created for the target
- An architecture & platform port of the 8051-cored CC2430
  - Porting the TinyOS core, the boot sequence and scheduler.
  - Creating drivers for the major subsystems of the CC2430
- A variant port of the chip’s radio
- Creating support for two sensorboards, involving the development of drivers for each device or sensor; allowing the functionality provided by these to be accessed by TinyOS applications.
  - RF04EB – Development Kit sensorboard
  - MARCO – Custom sensorboard

6.2 Wireless Sensor Networks

Due to recent advances in Micro-Electro-Mechanical Systems (MEMS) and ultra low-power processors Wireless Sensor Networks (WSNs) have become a reality. This area of computing has enjoyed substantial research efforts in recent past. It presents many new challenges and enables previously unviable, even impossible, applications to be developed. Saffo, of the Institute for the Future, made the following 10-year forecast in 1997 [1]:

*Just as the personal computer was a symbol of the '80s, the symbol of the '90s is the World Wide Web. Well, the next shift, the next nonlinear shift, is going to be the advent of cheap sensors.*

Potential application domains identified by [2] include:

- Military
- Environmental monitoring
- Health monitoring
- Home automation and monitoring
- Commercial - automation and monitoring

A wireless sensor network consists of a number of intelligent nodes, between tens and thousands but potentially millions, distributed throughout an area. The nodes generally inter-communicate using radio
frequencies (RF), although other wireless mediums such as optical are occasionally used for certain applications.

The highly-constrained resources of nodes, and unattended autonomous nature of applications, demand a new breed of operating system designed with the requirements of WSNs in mind. The typical execution cycle of a WSN application is presented in Figure 1 below. An application will generally have an extremely low duty-cycle (~1%).

Energy is a WSN device’s most precious resource. This affects both the hardware and software used; the hardware platform should have a minimal wakeup time to reduce wasting energy, and software should allow the device to spend as much time as possible in a low-power sleep mode.

Figure 1: Orthodox Lifecycle of WSN Applications

The scope and diversity of WSN applications is large, resulting in mote hardware, its capabilities and characteristics, being diverse also; [2] and [3] try to characterise the various types. A selection of WSN applications is presented in Table 1.

<table>
<thead>
<tr>
<th>Project</th>
<th>Purpose</th>
<th>Deployment &amp; Mobility</th>
<th>Required Lifetime</th>
<th>Number of nodes in Network</th>
<th>Network topology</th>
<th>Required node size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Great Duck Island</td>
<td>Habitat monitoring of bird nesting behaviour.</td>
<td>Manual deployment / Fixed nodes</td>
<td>7 months</td>
<td>Hundreds</td>
<td>Star mesh</td>
<td>Match-box sized</td>
</tr>
<tr>
<td>SESAME</td>
<td>Body monitoring of elite athletes</td>
<td>Manual deployment / Fixed nodes</td>
<td>Minutes / hours</td>
<td>A few per athlete</td>
<td>Star</td>
<td>Must be as small as possible to reduce impact on athlete performance.</td>
</tr>
<tr>
<td>Tracking</td>
<td>Tracking military vehicles</td>
<td>Random deployment from aircraft/ Mobile nodes</td>
<td>Potentially years</td>
<td>Hundreds</td>
<td>Mesh</td>
<td>Must be tiny to prevent detection by enemy</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------</td>
<td>---------------------------------------------</td>
<td>-------------------</td>
<td>----------</td>
<td>-----</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>Power Monitoring</td>
<td>Monitor power consumption in large and dispersed office buildings</td>
<td>Manual deployment / Fixed nodes</td>
<td>Years, but can use a power-grid</td>
<td>Hundreds</td>
<td>Star-mesh</td>
<td>Match-box sized</td>
</tr>
<tr>
<td>Parts Assembly</td>
<td>assist people during the assembly of complex composite objects such as do-it-yourself furniture</td>
<td>Manual deployment / Mobile nodes</td>
<td>Hours</td>
<td>Tens</td>
<td>Mesh</td>
<td>Unobtrusively small</td>
</tr>
<tr>
<td>ARGO</td>
<td>Ocean water monitoring</td>
<td>Random / Mobile nodes</td>
<td>Years</td>
<td>Thousands</td>
<td>Star</td>
<td>Shoe-box sized</td>
</tr>
</tbody>
</table>

Table 1: Comparison of WSN Applications & their Implications on the Hardware Platform

7. Network Topologies & Medium Access Control

The ways in which the nodes communicate with one another varies depending on the protocols and network topologies used. Two common schemes are collection, which routes data to the root of a tree topology, and dissemination, which reliably delivers a piece of data to every node in the network.

Due to nodes operating unattended, and being potentially inaccessible, WSNs utilise ad-hoc mesh networking. Mesh networks can either be fully connected, where each node can communicate directly with every other node, or partially connected (Figure 2), where multi-hop routing allows communication with any other node. They allow for continuous connections and reconfiguration around broken or blocked paths by hopping from node to node until the destination is reached. The nodes in a mesh network keep track of which devices are connected and provide intelligent routing in order to consume the least possible power; each node has the same routing capabilities.

- This offers redundancy, allowing the network to self-heal should a node fail or expend all its available power.
- In WSNs with random node deployment mesh networking is essential for establishing a path to the destination.
- When the short transmission range of typical nodes is ~30 metres, multi-hop routing is necessary to reach a distant base station designated as the root collection point.
- Mesh networking enables additional nodes deployed at a later date to join the network; creating greater density coverage or to replenish non-functioning nodes, for example.
- Adding more radios to a specific area creates the potential for higher data throughput, but the compromise is greater complexity and power consumption

![Figure 2: Mesh Network Topology - Partially Connected](image)

Variations of mesh network topologies are employed by some applications. Star-mesh shown in Figure 3 has redundant routing capabilities at the network core and star connected edge nodes. This allows less capable devices to be utilised, but requires a direct connection to one of the routing nodes.
The use of star network topologies is largely limited to PAN (Personal Area Network) applications, where there is a known coordinating device which is can be directly communicated with by all nodes; it must be accessible in a single hop.

![Network Topologies Employed by WSNs](image)

WSNs have contradictory communication aims; they should be responsive at any time, but should be energy efficient to the largest degree. As a result a large number of energy efficient medium access control (MAC) algorithms have been designed for WSNs with specific target purposes; algorithms include Aloha, S-MAC, T-MAC and the 802.15.4 MAC.

S-MAC [5], for example, consumes 2-6 times less energy than an 802.11-like MAC protocol. It requires nodes to be organised in clusters, and that all nodes are time synchronised. A large time interval is divided into an active listening period and a sleeping period. All nodes in the cluster are woken to listen at the same time.

Whilst Aloha with pre-amble sampling [6] ensures nodes do not listen, expending energy, when there is no transmission to receive. This algorithm causes the transmitting node to consume greater power however, as it must transmit a long pre-amble so that other nodes have time to wakeup before transmitting the data. This MAC algorithm is not as efficient as S-MAC, but does not require time synchronisation of nodes.

8. Wireless Standards

There are numerous wireless standards in existence, which cater for a number of application domains. Of these there is a subset of suitable candidates for WSNs. The key characteristics that these must process are low power consumption, low processor overhead, and small memory requirements, for their associated communications stack. Technologies such as WiFi are too power hungry and require a large stack (Table 2).

<table>
<thead>
<tr>
<th>Standard</th>
<th>Targeted Purpose</th>
<th>Theoretical Data Transfer Rate</th>
<th>Typical Stack Size</th>
<th>Typical Current Consumption (Tx/Rx)</th>
<th>Typical Current Consumption (Sleep)</th>
</tr>
</thead>
<tbody>
<tr>
<td>802.15.4</td>
<td>Generic</td>
<td>250kbps</td>
<td>&lt; 64kbyte Potentially substantially less depending on the upper layers.</td>
<td>~18mA</td>
<td>~5μA</td>
</tr>
<tr>
<td>(2.4GHz)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WiBree</td>
<td>Personal Area Networks (PANs)</td>
<td>1Mbps</td>
<td>Unknown</td>
<td>Standalone chip expected to consume ~15mA</td>
<td>Standalone chip expected to consume ~5μA</td>
</tr>
<tr>
<td>Bluetooth 1.2</td>
<td>Cable replacement</td>
<td>1Mbps</td>
<td>&gt; 250kbyte</td>
<td>~30mA</td>
<td>~6mA</td>
</tr>
<tr>
<td>WiFi</td>
<td>Computer networking</td>
<td>54Mbps (802.11g)</td>
<td>&gt; 1000kbyte</td>
<td>~500mA</td>
<td>~10mA</td>
</tr>
</tbody>
</table>
Table 2: Comparison of Wireless Standards [7] [10] [13] [14]

It can be seen in Table 2 that the 802.15.4 standard [8] is particularly well suited to WSNs. Bluetooth still has a place in WSN applications, though at a higher tier in the class hierarchy (see Section 9.2). Its larger bandwidth can be utilised by data-collection nodes acting as proxies, which have less power constraints. Additionally, in this scenario Bluetooth’s limited number of possible connections and lack of multihop capabilities [9] is not as great a problem. It should also be noted that it is possible to create a highly stripped-down Bluetooth stack, with a size 1/25th that quoted in Table 2 [10].

A number of early WSN platforms were equipped with Bluetooth radios, such as the BT Node [11] and Imote 1.0 [12], however the availability of 802.15.4 radios has largely superseded its use. For example, the Imote 2.0 [49] employs an 802.15.4 compliant radio.

Little is currently known about WiBree [13], which is being developed by Nokia and Nordic Semiconductor amongst others, as only press-release information is available. However, it appears to be a promising alternative to 802.15.4 for certain application types [14]. It has a short transmission range (10 metres), which largely limits its suitability to Personal Area Networks (PANs) and Body Sensor Networks (BSNs) type applications; this is its target purpose.

802.15.4 is important, as it provides a standard for interoperable radio communication, which satisfies the requirements of WSNs. Specifically it has been designed for low power, low data, high density wireless networks; previous RF technologies which have exhibited these properties have not been standardised. However, it should be noted that the 802.15.4 support in TinyOS is not fully-featured. it only provides ad-hoc communication [15] attempted to implement a full 802.15.4 MAC layer, though found the timing requirements prevented success. However, the radio hardware available does not strictly comply with the standard, as the silicon vendors have acknowledged shortfalls with it [16]; Flora and Bonnet have capitalised on this and, by not strictly complying with the standard have successfully implemented a full 802.15.4 MAC layer [16].

The 802.15.4 standard [8] specifies data-link and physical layers of the stack allowing higher layers to be implemented in a suitable way for the target purpose; TinyOS [17] and ZigBee [18] are examples of this. The ZigBee standard [19] defines a number of network topologies and profiles targeted at industrial and home automation. Its relationship to 802.15.4 is shown in Figure 4.

Figure 4: Relationship of ZigBee to 802.15.4 [20]


Romer and Mattern [3] acknowledge that WSNs have been utilised in a wide variety of problem domains and, as a result, systems have vastly varying requirements and characteristics. Their survey offers evidence that there is no unified solution, where a single platform is suitable for all applications. Additionally Beutel [21] attempts to compare the metrics of various WSN platforms, but concludes that it is difficult to compare platforms due to the diverse scope of applications and that each platform has its own distinct characteristics. However, some general characteristics can be observed; a comparison of a number of nodes’ properties is presented in Section 9.1.

Sensor nodes, known as motes, typically have highly constrained resources. While the nature of WSN applications dictates high levels of concurrency; sensor sampling, packet routing, data transmission and reception, and event handling must all be performed concurrently. The result of this is the impact on energy efficiency, code size and RAM usage of applications must be taken into serious consideration.

Nicholas Beck  
Student Number: 03971477
A sensor node’s most precious resource is energy. WSN applications may require nodes to run unattended for years, due to their inaccessible location or cost of deployment. Furthermore, the choice of radio and processor are one of the key aspects of the platforms ultimate energy efficiency.

Lynch and O’Reilly [30] identify a number of factors which affect processor suitability for use in WSN platforms. In determining the power used by sensing application it is important to consider the typical execution cycle (Figure 1). The result of the low duty-cycle “timer-based state machine” approach results in MCU wakeup time being one of the most important consideration, as the period before useful instructions are executed is wasted energy. Additionally, if a MCU has an ultra-quick startup time it is possible to put the device into a low-power mode between the start and completion of a split-phase operation.

Additionally, [30] states that the greater the flexibility of power-saving nodes and clocks provided by the MCU, the greater the power efficiency that can generally be realised. The ISA (Instruction Set Architecture) and memory architecture of the device can also amount to substantial energy differences. A more complex ISA often allows a device to execute the event handler in fewer instructions, allowing it to return to a low-power mode sooner. Whilst the type of memory, and memory access instructions, can introduce instruction overhead and therefore power dissipation.

### 9.1 Comparison of Available Mote Platforms

Table 3 below compares a number of available WSN platforms from various classes; Section 9.2 contains a discussion of device classes. The CC2430 is the same device class as the MicaZ and Telos motes, which are the de facto research platforms. The CC2430 compares favourably with other motes of its class. Furthermore, it offers cost, reliability and size benefits because it is a SoC device. This is discussed in greater detail in this section.

<table>
<thead>
<tr>
<th>Mote</th>
<th>Class &amp; Type</th>
<th>CPU</th>
<th>MCU &amp; Radio Wakeup</th>
<th>Current Consumption (MCU+Radio)</th>
<th>Radio</th>
<th>Memory</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micaz</td>
<td>Generic sensing platform</td>
<td>Atmega128</td>
<td>MCU: 180µs</td>
<td>33mA active 30µA idle</td>
<td>CC2420</td>
<td>4K RAM 128K Flash</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unintegrated</td>
<td>Radio: 860µs</td>
<td></td>
<td></td>
<td>250kbps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Telos B</td>
<td>Generic sensing platform</td>
<td>MSP430</td>
<td>MCU: 6µs</td>
<td>22mA active 5µA idle</td>
<td>CC2420</td>
<td>10K RAM 48K Flash</td>
<td>Can operate at 1.8V</td>
</tr>
<tr>
<td></td>
<td>Unintegrated</td>
<td>Radio: 580µs</td>
<td></td>
<td></td>
<td>250kbps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CC2430</td>
<td>Generic sensing platform</td>
<td>8051</td>
<td>MCU: 89µs</td>
<td>27mA active 0.5µA idle</td>
<td>Integrated  CC2420 250kbps</td>
<td>8K RAM 32/64/128K Flash</td>
<td>Can operate at 2.0V</td>
</tr>
<tr>
<td></td>
<td>Integrated</td>
<td>Radio: 320µs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spec</td>
<td>Specialised sensing</td>
<td>Custom 8-bit</td>
<td></td>
<td>3mA active 3µA idle</td>
<td>50-100kbps</td>
<td>UHF radio</td>
<td>3K RAM</td>
</tr>
<tr>
<td></td>
<td>platform</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iMote 2.0</td>
<td>High computational-power</td>
<td>Xscale - PXA271</td>
<td></td>
<td>66mA active 390µA idle</td>
<td>CC2420</td>
<td>256K SRAM 32MB Flash</td>
<td>Max. clock speed = 104MHz</td>
</tr>
<tr>
<td></td>
<td>platform</td>
<td></td>
<td></td>
<td>At 104MHz.</td>
<td>250kbps</td>
<td>32MB SDRAM</td>
<td>Min. clock speed = 13MHz</td>
</tr>
</tbody>
</table>

Table 3: Comparison of Available WSN Hardware
9.2 Classes of Mote

Hill et al. identified that virtually all WSNs employ a tiered architecture [22], consisting of a number of mote classes, as shown in Figure 5, whose approximate capabilities are shown in Table 4. The author has observed that, to date, virtually all WSNs have utilised two functional levels: a sensor network, consisting of a number of motes, and a data collection server. However, from an assessment of WSN research currently being undertaken there is little evidence of multi-tiered wireless sensor networks as depicted in Figure 5. Surveys of WSN projects undertaken by others have yielded similar results [2], [3]. BIONETS [23] uses two node classes, high computational-power platform and generic sensing platform, however there is no communication between the lower-level nodes, which is moving away from conventional sensor networks; the two classes effectively act as master and slave respectively.

To date WSN research has been predominantly undertaken using generic sensing platforms. However, the author anticipates WSN applications will employ multiple tiers in the near future.

At the time of writing there are very few specialised motes that have been fabricated (SPEC [24, Ch.5]) or are being developed [25], which prevents the lowest tier from being utilised. It should be noted that all specialised motes are currently research projects; they are not purchasable.

In the recent past an acknowledgement and appreciation of the need for more powerful motes, with respect to both computational and data transfer capabilities, has occurred. A number of 32-bit platforms with large memories and higher clock speeds have been created, such as the Jennic JN5139-000 and the Crossbow Imote2. This class of mote will likely see heavy usage as “super-nodes”; they are capable of performing useful data processing and can buffer large amounts of data enabling them to act as data gateways. The power consumption of this class of mote is approximately an order of magnitude greater as a consequence.

Figure 5: Classes of Mote Platforms [22]
9.3 Types of Mote

9.3.1 Common Off-The-Shelf (COTS) Motes

9.3.1.1 Unintegrated

The vast majority of COTS motes currently available are unintegrated, MCU plus radio, designs; a selection of which is shown are Figure 6. These hardware platforms utilise discrete components which does not lead to the most energy efficient solution. Whilst improvements have been made over the first platforms, by motes such as the MicaZ which utilises an 802.15.4 radio, they are really generation 1.5 devices.

Additionally, these devices are relatively large (roughly the size of a matchbox), which prevents them from being suitable for certain applications, where weight and size are critical. The SESAME project [26], which monitors elite athletes, has suffered complaints from athletes about the device’s weight. Also, the devices’ size prevents their use in military surveillance, where they would be easily discovered and destroyed.

![Figure 6: A Selection of COTS Motes: (left to right) BT Node, Mica2, Tmote Sky, Sun SPOT](image)

9.3.1.2 System-on-Chip

A few SoC devices, such as the Chipcon CC2430 (Figure 7) and the Jennic JN5139-000, have recently become available. These offer substantial benefits; they are cheaper, they can be deployed in extremely small footprint devices, have a reduced complexity for their circuit design, and potentially an ultra-low power consumption. These devices represent the second generation of WSN devices.

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Sample “Name” and Size</th>
<th>Typical Application Sensors</th>
<th>Radio Bandwidth (Kbps)</th>
<th>MIPS Flash RAM</th>
<th>Typical Active Energy (mW)</th>
<th>Typical Sleep Energy (µW)</th>
<th>Typical Duty Cycle (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specialized sensing platform</td>
<td>Spec mm³</td>
<td>Specialized low-bandwidth sensor or advanced RF tag</td>
<td>&lt;50Kbps</td>
<td>5</td>
<td>1.8V&lt;10–15mA</td>
<td>1.8V&lt;1uA</td>
<td>0.1–0.5%</td>
</tr>
<tr>
<td>Generic sensing platform</td>
<td>Mote 10cm³</td>
<td>General-purpose sensing and communications relay</td>
<td>&lt;100Kbps</td>
<td>10</td>
<td>3V&lt;10–15mA</td>
<td>3V&lt;10µA</td>
<td>1–2%</td>
</tr>
<tr>
<td>High-bandwidth sensing</td>
<td>Imote 10cm³</td>
<td>High-bandwidth sensing (video, acoustic, and vibration)</td>
<td>~500Kbps</td>
<td>50</td>
<td>3V&lt;60mA</td>
<td>3V&lt;100µA</td>
<td>5–10%</td>
</tr>
<tr>
<td>Gateway</td>
<td>Scargate &gt;10cm³</td>
<td>High-bandwidth sensing and communications aggregation Gateway node</td>
<td>&gt;500Kbps–10 Mbps</td>
<td>100</td>
<td>3V&lt;200mA</td>
<td>3V&lt;10mµA</td>
<td>&gt;50%</td>
</tr>
</tbody>
</table>

Table 4: Typical Characteristics & Properties of Mote Classes [22]
9.3.2 Custom Hardware Motes

Custom hardware designs, such as SPEC (Figure 8), represent the future of specialist sensing platforms. These devices are highly specialised for WSNs. Specifically, SPEC [24] has an additional set of registers dedicated for interrupt handlers. Instead of saving the current execution context, a single instruction in the ISA (Instruction Set Architecture) is executed to switch the register set window. Considering the event-triggered nature of WSN applications, and that wakeup time is one of the most critical aspects of energy efficiency [30], this provides substantial gains, which can be translated to better runtime performance or greater energy efficiency.

![Figure 8: The SPEC Mote pictured next to a ballpoint pen](image)

Hempstead et al. [25], have moved away from the traditional MCU in favour of an asynchronous processor. This is still being developed, however, being event-driven should offer substantial energy efficiency improvements due to regular events being handled by a hardware state machine.

PAWiS [27], acknowledges that currently available COTS SoC motes do not take optimisation of the overall device into consideration. Specifically, its generic architecture prevents their use for specialised sensing. They also propose the use of a true wakeup receiver. Current platforms expend a considerable amount of energy listening for data transmissions; listening for data transmissions draws approximately the same current as transmission. Whilst the development of energy efficient MAC algorithms attempts to combat this, they are no substitution for an ultra-low power listening mode. However, there is no efficient implementation available.

9.4 The Chipcon CC2430 Mote

The Chipcon CC2430 [28] is a System-on-Chip mote that consists of a number of functional blocks: the two main components are the 8051 processor core and radio transceiver, plus a large number of peripheral components, as shown in Figure 9 and listed below.

- Enhanced 8051 processor core
- 802.15.4 compliant 2.4GHz radio transceiver
- 21 GPIO pins
- Shared with peripherals
- 2 USARTs
- Each configurable as either a SPI or UART controller
- 5 timers
- 1 x 16-bit
- 1 x MAC timer (High-precision 16-bit timer with 20-bit overflow counter)
- 2 x 8-bit timers
- Sleep timer (24-bit)
- 13-bit A/D Converter
- Random number generator
- AES coprocessor
- Power management controller
- Watchdog timer (WDT), Power On Reset (POR) & Brown Out Detector (BOD)

Figure 9: CC2430 Block Diagram [28]

The CC2430 is an example of a 2nd generation, integrated, mote. It is a generic sensing platform class mote as described in Section 9.2.

The CC2430 is a System-on-Chip (SoC) device, allowing for a smaller footprint than comparable un-integrated (MCU plus radio) motes. For example the standalone CC2420 radio chip has a 6x6mm² footprint and requires a separate MCU, whereas the SoC CC2430 combines the MCU and radio into a single 7x7mm² package.

Due to the lower number of solder joints on the integrated package the probability of failure is decreased and reliability increased. This also reduces the bill-of-materials (BOM) and makes the PCB design simpler, reducing the cost of fabricating the motes.

Peripherals that provide hardware optimisations for certain tasks, such as the DMA controller and AES coprocessor, help increase energy efficiency by reducing load on the processor. For example, a channel of the DMA controller can be initialised allowing a data transfer to take place whilst the processor is in a low-power sleep mode.

Specific considerations for individual peripherals, with respect to design decisions and suitability for TinyOS, are presented in Section 11.
9.4.1 Processor Core

The CC2430 contains an enhanced 8051 processor core. [29] identifies the 8051 processor-core as a potential candidate for WSN platforms, and has been chosen by Chipcon and Nordic Semiconductor as the MCU core for their SoC offerings.

The CC2430 employs an enhanced, low-power, 8051 processor-core which has a number of sleep modes that allow for ultra-low-power operation, for example wake on interrupt or real-time clock draws 500μA at 3V (Table 5, Power mode 2). The various power modes allow a compromise to be made between current consumption and wake-up time (Table 5 and Table 6), making the CC2430 suitable for WSN applications that either have an ultra-low or moderate duty cycle.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Min</th>
<th>Typ</th>
<th>Max</th>
<th>Unit</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Current Consumption</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCU Active Mode, 16 MHz</td>
<td>4.3</td>
<td>mA</td>
<td></td>
<td></td>
<td>Digital regulator on, High frequency (16 MHz) RCOSC running.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No radio, crystals, or peripherals active. Code run with Cache hit.</td>
</tr>
<tr>
<td>MCU Active Mode, 32 MHz</td>
<td>9.5</td>
<td>mA</td>
<td></td>
<td></td>
<td>MCU running at full speed (32MHz), 32MHz XOOSC running. No radio or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>peripherals active. Code run with Cache hit.</td>
</tr>
<tr>
<td>MCU Active and RX Mode</td>
<td>26.7</td>
<td>mA</td>
<td></td>
<td></td>
<td>MCU running at full speed (32MHz), 32MHz XOOSC running, radio in RX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mode, -50 dBm input power. No peripherals active. Code run with Cache</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>hit.</td>
</tr>
<tr>
<td>MCU Active and TX Mode, 0dBm</td>
<td>26.9</td>
<td>mA</td>
<td></td>
<td></td>
<td>MCU running at full speed (32MHz), 32MHz XOOSC running, radio in TX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mode, 0dBm output power. No peripherals active. Code run with Cache hit.</td>
</tr>
<tr>
<td>Power mode 1</td>
<td>190</td>
<td>μA</td>
<td></td>
<td></td>
<td>Digital regulator on, High frequency RCOSC and crystal oscillator off.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32,768 kHz XOOSC, POR and ST active. RAM retention.</td>
</tr>
<tr>
<td>Power mode 2</td>
<td>0.5</td>
<td>μA</td>
<td></td>
<td></td>
<td>Digital regulator off, High frequency RCOSC and crystal oscillator off.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32,768 kHz XOOSC, POR and ST active. RAM retention.</td>
</tr>
<tr>
<td>Power mode 3</td>
<td>0.3</td>
<td>μA</td>
<td></td>
<td></td>
<td>No clocks. RAM retention. POR active.</td>
</tr>
</tbody>
</table>

Table 5: CC2430 Current Consumption in Various Power Modes

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Min</th>
<th>Typ</th>
<th>Max</th>
<th>Unit</th>
<th>Condition/Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Wake-Up and Timing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power mode 1 → power mode 0</td>
<td>4.1</td>
<td>μs</td>
<td></td>
<td></td>
<td>Digital regulator on, High frequency RCOSC and crystal oscillator off.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Start-up of High frequency RCOSC.</td>
</tr>
<tr>
<td>Power mode 2 or 3 → power mode 0</td>
<td>69.2</td>
<td>μs</td>
<td></td>
<td></td>
<td>Digital regulator off, High frequency RCOSC and crystal oscillator off.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Start-up of regulator and High frequency RCOSC.</td>
</tr>
<tr>
<td>Active → TX or RX</td>
<td>525</td>
<td>μs</td>
<td></td>
<td></td>
<td>Time from enabling radio part in power mode 0, until TX or RX starts.</td>
</tr>
<tr>
<td>32MHz XOOSC initially OFF. Voltage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Includes start-up of voltage regulator and crystal oscillator in parallel.</td>
</tr>
<tr>
<td>regulator initially OFF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Crystal ESR = 16Ω.</td>
</tr>
<tr>
<td>Active → TX or RX</td>
<td>320</td>
<td>μs</td>
<td></td>
<td></td>
<td>Time from enabling radio part in power mode 0, until TX or RX starts.</td>
</tr>
<tr>
<td>Voltage regulator initially OFF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Includes start-up of voltage regulator.</td>
</tr>
<tr>
<td>Active → RX or TX</td>
<td>192</td>
<td>μs</td>
<td></td>
<td></td>
<td>Radio part already enabled. Time until RX or TX starts.</td>
</tr>
<tr>
<td>RX/TX turnaround</td>
<td>192</td>
<td>μs</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Wake-up Times from Various Power Modes

The CC2430’s 8051 core has a greater wake-up time than some other low-power processor cores [30] (Table 3), which when the general lifecycle of a WSN application is considered (shown in Figure 1) makes the CC2430 a less attractive option. However, with the advent of small package, high-density, super-capacitors (Aerogel B1835-2R5336-R: 50F capacitor in a 18x35mm package [31]) and cheap solar cells this is somewhat overcome; as demonstrated by Jiang et al. [32], who show that a low-duty-cycle application has the capability of operating for decades. Furthermore, the combined MCU and
radio startup time is faster than the Telos, so in applications which utilise the radio every duty-cycle the wasted energy during startup is comparable.

Additionally, the enhancements to the 8051 core decrease the number of machine cycles required to execute an instruction, typically giving eight times better performance than the standard core [28], which increases the period the chip can remain in a low power mode.

9.4.2 Radio Transceiver

The CC2430 employs the 2.4GHz 802.15.4 compliant CC2420 radio transceiver, which is also available as a discrete IC from Chipcon. From Table 7 it can be seen that the CC2420 significantly outperforms other radios in its class.

<table>
<thead>
<tr>
<th>Type</th>
<th>Narrowband</th>
<th>Wideband</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>8051</td>
<td>CC2420</td>
</tr>
<tr>
<td>Part no.</td>
<td>TR1000</td>
<td>CC1000</td>
</tr>
<tr>
<td>Max Data rate (kbit/s)</td>
<td>115.2</td>
<td>96.8</td>
</tr>
<tr>
<td>RX power (mW)</td>
<td>3.8</td>
<td>9.6</td>
</tr>
<tr>
<td>TX power (mW/4dBm)</td>
<td>12 / 1.5</td>
<td>16.5 / 10</td>
</tr>
<tr>
<td>Forward power (μA)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Turn on time (μs)</td>
<td>0.02</td>
<td>2.1</td>
</tr>
<tr>
<td>Modulations</td>
<td>OOK/ASK</td>
<td>FSK</td>
</tr>
<tr>
<td>Packet detection</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Address decoding</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Encryption support</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Error detection</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Error correction</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Interface</td>
<td>bit</td>
<td>byte</td>
</tr>
<tr>
<td>Buffering (bytes)</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>Time-sync</td>
<td>bit</td>
<td>SFD/byte</td>
</tr>
<tr>
<td>Localization</td>
<td>ESSI</td>
<td>RSSI</td>
</tr>
</tbody>
</table>

* Manufacturer's documentation does not include additional information.

Table 7: Comparison of Radio Chips Suitable for WSN Platforms [33]

The integrated 802.15.4 compatible CC2420 radio operates in the 2.4GHz band, which requires larger transmission power for the same range in comparison to VHF and UHF radio bands. However, this is offset by the greater bandwidth it possesses allowing faster data transmission, which results in a lower energy expenditure per-bit, due to the reduced period that the MCU and radio transceiver must remain active.

9.5 Evaluation of the CC2430 as a Mote Platform

The Chipcon CC2430 has a number of characteristics that make it a suitable device for use in WSN applications. It has a small footprint which requires few external components resulting in a low Bill-Of-Materials (BOM), which fulfils two of the main required characteristics of a mote platform. The third required characteristic, energy efficiency, is satisfied through a number of factors discussed above.

The use of an 802.15.4 radio enables the CC2430 to interoperate with a number of other platforms in heterogeneous networks; such as the TelosB, MicaZ and Imote 2.0.

It can be seen from Table 3, that the CC2430 appears to be an ideal candidate for WSN platforms, from looking at its datasheet. It is a direct contender with the Telos, which performs significantly better than the MicaZ. However the CC2430 offers the benefits of a SoC device discussed above.

10. Operating Systems for Wireless Sensor Networks

Traditional embedded operating systems such as VxWorks, WinCE, QNX and uCLinux are not suitable for WSNs. These often do not scale down to 8-bit MCUs. Their memory requirements are orders of magnitude greater than available. Their thread-based POSIX programming models result in context switches that amount to a large overhead, which does not suit the typical execution cycle (Figure 1) of WSN applications.

Thread-based embedded operating systems do not suit the event-driven nature of WSNs, which typically have a duty-cycle of 1% [30]; maintaining low-power operation for the greatest possible...
period is essential to reduce energy consumption. The properties of a number of embedded operating systems and real-time executives are presented in Table 8.

WSN applications require an efficient software architecture that allows high levels of concurrency and provides optimal energy efficiency.

<table>
<thead>
<tr>
<th>Name</th>
<th>Preemption</th>
<th>Protection</th>
<th>ROM Size</th>
<th>Configurable</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>pOSEK</td>
<td>Tasks</td>
<td>No</td>
<td>2K</td>
<td>Static</td>
<td>Microcontrollers</td>
</tr>
<tr>
<td>pSOSSystem</td>
<td>POSIX</td>
<td>Optional</td>
<td>Dynamic</td>
<td>Dynamic</td>
<td>Strong ARM, ARM Thumb</td>
</tr>
<tr>
<td>VxWorks</td>
<td>POSIX</td>
<td>Yes</td>
<td>≈ 286K</td>
<td>Dynamic</td>
<td>Pentium</td>
</tr>
<tr>
<td>QNX Neutrino</td>
<td>POSIX</td>
<td>Yes</td>
<td>&gt; 100K</td>
<td>Dynamic</td>
<td>Intel, NEC chips</td>
</tr>
<tr>
<td>QNX Realtime</td>
<td>POSIX</td>
<td>Yes</td>
<td>100K</td>
<td>Dynamic</td>
<td>Pentium II → 386's</td>
</tr>
<tr>
<td>OS-9</td>
<td>Process</td>
<td>Yes</td>
<td>Dynamic</td>
<td>Dynamic</td>
<td>Pentium → SH14</td>
</tr>
<tr>
<td>Chorus OS</td>
<td>POSIX</td>
<td>Optional</td>
<td>10K</td>
<td>Dynamic</td>
<td>Pentium → Strong ARM</td>
</tr>
<tr>
<td>Ariel</td>
<td>Tasks</td>
<td>No</td>
<td>19K</td>
<td>Static</td>
<td>SH2, ARM Thumb</td>
</tr>
<tr>
<td>CREEM</td>
<td>data-flow</td>
<td>No</td>
<td>560 bytes</td>
<td>Static</td>
<td>ATMEL 8051</td>
</tr>
</tbody>
</table>

Table 8: Comparison of Embedded Operating Systems [34]

10.1 Comparison of Wireless Sensor Network Operating Systems

There are a number of WSN operating systems in existence, which have been designed specifically for this domain; a selection of which is presented in Table 9.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Architecture Type</th>
<th>Multimodal Tasking</th>
<th>Dynamic Reprogramming</th>
<th>Priority-based Scheduling</th>
<th>Real-time Guarantee</th>
<th>Programming Language</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TinyOS</td>
<td>Component based.</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>nesC</td>
<td>Static component instantiation. Full static analysis.</td>
</tr>
<tr>
<td>MANTIS [35], [36]</td>
<td>Thread based.</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>C</td>
<td>Supports pre-emptive multithreading.</td>
</tr>
<tr>
<td>Contiki [37], [38], [39], [40]</td>
<td>Event-driven kernel. Plus preemptive multithreading provided at user-level by library</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>C</td>
<td>Blocking call semantics provided by protothreads, which evaluate to non-blocking state machines.</td>
</tr>
<tr>
<td>NanoQplus [41], [42]</td>
<td>Thread based.</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>C</td>
<td>Utilises a nano-kernel, provides a POSIX-like API. Provides socket-like communication abstractions.</td>
</tr>
<tr>
<td>Nano-RK [43], [44]</td>
<td>Thread-based.</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>C</td>
<td>Static task control block allocation</td>
</tr>
<tr>
<td>SOS [45], [46]</td>
<td>Module based.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>C</td>
<td>Primary motivation is dynamic reprogramming. A common kernel provides messaging, dynamic memory, module (un)loading amongst other services. Modules are self-contained and cooperatively scheduled, but do not provide memory protection like processes.</td>
</tr>
</tbody>
</table>

Table 9: Comparison of WSN Operating Systems
Event-driven systems provide concurrency without the need for per-thread stacks or locking mechanisms [47]. Processes are implemented as event handlers that run to completion. Because an event handler cannot block, the limited memory resources can be shared by all processes.

All thread-based approaches result in larger memory requirements. While it has been shown that this does not present problems with generic-sensing platform class motes [36], which are currently used at large, portability to extremely resource constrained specialist sensing class devices may not be possible.

To date, the vast majority of WSN research has employed TinyOS, and its simulator TOSSIM; TinyOS is discussed in detail in Section 10.2.

Contiki employs an event-driven kernel, but supports pre-emptive multi-threading through a user-space library. Additionally, it provides blocking semantics through the use of protothreads [39]. Protothreads, in their macro function form, are evaluated at compilation where their blocking semantics style implementation is transformed into a non-blocking state machine; this allows a more traditional programming paradigm to be used.

TinyMOS [48] claims TinyOS can be run as one of MANTIS’ execution threads, in much the same way as RTAI does with Linux. This offers real-time and pre-emptive multi-threading capabilities to TinyOS applications. The application can delegate a particular task that requires this functionality to MANTIS in order to guarantee deadlines are met.

From information in the public domain it appears the use of NanoQplus is isolated to its creators. It employs a pre-emptive scheduler in order to provide the real-time capabilities it focuses on providing. While the term “real-time” is subject to much debate, and refers to the consistency of response times to events, rather than the timeliness of responses themselves, there is an element of timeliness implied. However, there are no stated figures for either consistency of response or timeliness.

It is arguable whether real-time capabilities are necessary for WSN applications; if it refers to the detection of a critical condition which requires immediate action, for example, there are significant issues. When a typical WSN application keeps the mote in a low-power sleep mode 99% of the time for energy efficiency reasons, real-time capabilities would have no effect. It would not be possible for the device that detected an event to transmit the data to a basestation for a number of potential reasons:

- Depending on the MAC protocol in use, the node may have to wait for the designated transmission period, or a long-preamble may be required.
- If multihop routing is required the device would have to wait for the routing node to wakeup and listen so that it could receive the packet.
- The network may be heavily loaded with traffic, preventing data from being transmitted for a significant period.

Nano-RK [44] argues the need for real-time capabilities from the perspective that local tasks have associated deadlines with respect to fulfilling end-to-end deadlines. The need for priority-based pre-emption is all discussed, with the opinion that the burden of ensuring a task is scheduled within a certain period should be on the OS, not the developer. They state that it becomes necessary when multiple inputs must be serviced at different rates, because of limited timer interrupts on low-end MCUs. While the points made are valid, emphasis is placed on applications with very time-consuming data processing on individual nodes; this may be a possibility in future applications, though is questionable considering the energy constraints of nodes. Furthermore, although “super-nodes” were described as performing data processing in Section 9.2, it should be appreciated that devices of this class have superior processing power and greater resources; thereby not suffering from the same problems as the low-end MCUs described by [44].

### 10.2 TinyOS

TinyOS [49] is an open-source operating system designed for wireless embedded sensor networks. It features a component-based event-driven architecture which allows application code size to be minimal, as required by the severe memory constraints inherent in sensor networks.

TinyOS is the de facto environment for wireless sensor network development. It has been coined as an operating system within the WSN community and literature at large, but in reality TinyOS is more
comparable to a real-time executive (RTE). Its emphasis is on reacting to external events and ultra-low power operation.

The core TinyOS system and TinyOS applications are written in a language called nesC (network embedded system C), which is discussed in Section 10.2.1.

Every resource and service is bound at compile time and all allocation is static. This is achieved through wiring, which generates a full-program component call graph, allowing static analysis to be performed. Specifically, TinyOS applications are consist of only the required components for the particular application, which are composed into a single executable. This approach provides semi-automatic configuration, as opposed to the often manual configuration of micro- and nano-kernels.

The event-driven architecture and associated concurrency model allow for a unified stack and no locking mechanisms. This is a great benefit for WSN devices as RAM is usually the limiting software resource [50]. The disadvantages of this approach are that the state driven programming model can be difficult to manage [51], where split-phase operations are utilised instead of blocking calls, and the potential for starvation [38]. The onus is placed on the developer to ensure a lengthy computation does not monopolise the CPU. Split-phase operations present software component like a piece of hardware, where an event is signalled when the operation completes.

10.2.1 NesC

The nesC language [52], used to implement TinyOS, has been developed from the ground-up for devices with extremely limited resources. It was developed to allow TinyOS to meet the requirements of WSNs. Its key focus is holistic system design [53].

NesC is a component-based C dialect. TinyOS components have some similarities to objects in that they encapsulate state and couple state with functionality, however there are also significant differences. The principle distinction lies in their naming scope. Specifically, they use a purely local namespace, which means that in addition to declaring the functions a component implements, it must also declare the functions that it calls [54]. Component signatures therefore contain interfaces that they provide, which the component implements, and use, which the component handles.

Additionally, components and their interactions are fixed at compile-time, to increase reliability and efficiency through static program analysis, rather than at run-time as object-oriented languages do.

Developing a nesC program involves writing components and wiring users to providers. The component composition occurs at compilation and, as a result, does not require runtime allocation or use RAM to store function pointers. Additionally, because a program does not have these levels of indirection, the nesC compiler knows the complete call graph.

The nesC compilation model allows full program analysis. This enables it to perform heavy optimization of the component compositions, which appear to be very inefficient. The nesC compiler often inlines a call that crosses five or six components boundaries into a flat instruction stream with no function calls [54].

An important design consideration for all subsystems is that named sub-parts will be included in the application [55]. The nesC compiler removes unreachable code that it can guarantee will not occur, but there are cases where it is impossible for the compiler to determine this, resulting in the inclusion of useless code. For example unused interrupt handlers are never removed and therefore all code reachable from them will be included in the TinyOS application.

10.2.1.1 Wiring

Interfaces define a bidirectional relationship between components: the downcall and upcall of a split-phase operation are syntactically bound together; this is termed wiring.

In order to call the downcall, a component must implement the upcall. Conversely, a component can only signal the upcall if it implements the downcall [50]. This programming paradigm has no analogue.

There are two forms of nesC wiring: uses -> provides, and pass-through (=) [54]. The -> operator explicitly wires users of an interface to providers of that interface, while the = operator allows wiring to be deferred to higher-level components amongst other things.
10.2.1.2 Attributes

NesC provides the ability to annotate functions, variables, and interfaces, to inform the compiler of required behaviour. This allows component composition errors to be caught at compile-time. For example, the `@atmostonce` attribute can be used to prevent usage assumption violations of an interface. This will force the compiler to generate an error if the application has wired to the interface more than once. Custom attributes can also be defined, which can be used to decorate C and nesC declarations and definitions; this information can be dumped for use in an external tool. For the full list of natively supported attributes see the nesC reference manual [56].

10.2.1.3 Compile-time Constant Functions

NesC currently has two compile-time constant functions, which are effectively macro functions. The functionality of `unique()` and `uniqueCount()`, described in [56], allow static binding to be performed. They are used for obtaining the number of instances that exist in an application for two purposes: static allocation of the necessary array sizes, for virtualised components, and the automatic wiring of parameterised interfaces to particular instances of the interface. Constant functions are discussed with examples for the subsystems which they affect below.

10.2.2 Static Analysis

Since WSN devices generally operate unattended, and may be located in an inaccessible location, it is critical that potential runtime errors are minimized at the time of compilation. Although a number of projects focused on creating in-field WSN debugging tools it is widely accepted that prevention is better than cure. This can be accomplished through static analysis of the application. Additionally, because the call graph for the entire program is known, aggressive optimization can be performed, helping to achieve the requirements of resource constrained devices.

NesC provides a number of mechanisms for increasing the reliability and efficiency of TinyOS applications [53], [54], [56]:

- All allocation and interactions are static, which allows the full call graph at compile-time, which leads to highly accurate program analysis.
- It reduces the expressive power and structure through components and interfaces.
- Attributes allow component usage assumptions to be checked for abuse at compile-time.
- Typed interfaces allow compile-time checking to be performed.
- Data-race detection is performed at compile-time
- Whole program inlining is performed, which results in heavy optimization and reduces the apparent overhead of a component-based architecture to an extent that it is insignificant. The level of optimization this performed is illustrated in Table 10 and Table 11
- Static bidirectional binding and the component-based architecture ensure that the program only consists of essential code, and therefore results in a minimal code size. For example the TinyOS core requires 400 bytes of code and data memory combined.

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Optimized</th>
<th>Unoptimized</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work</td>
<td>371</td>
<td>520</td>
<td>29%</td>
</tr>
<tr>
<td>Boundary crossing</td>
<td>109</td>
<td>258</td>
<td>57%</td>
</tr>
<tr>
<td>Non-interrupt</td>
<td>8</td>
<td>194</td>
<td>93%</td>
</tr>
<tr>
<td>Interrupt</td>
<td>101</td>
<td>64</td>
<td>36%</td>
</tr>
<tr>
<td>Total</td>
<td>480</td>
<td>778</td>
<td>38%</td>
</tr>
</tbody>
</table>

Table 10: Effect of inlining on application footprint and performance. All sizes are in bytes. The CPU reduction column shows the reduction in CPU cycles spent executing tasks with optimizations enabled [53].
Table 11: Optimization effects on timer event handling. This figure shows the breakdown, in CPU cycles, for both work and boundary crossing for timer event handling, which requires 7 module crossings. Optimization reduces the overall cycle count by 38% [53].

### 10.2.3 Components & Interfaces

TinyOS applications consist of components, which can be either modules or configurations, and interfaces.

#### 10.2.3.1 Interfaces

Interfaces define sets of operations, as shown in Code Sample 1. The operations can be either commands, which are called, or events, which are signalled. Components either use or provide interfaces. When a component provides an interface it must implement its command operations, and if a component uses an interface it must implement all of its event handlers.

```cpp
interface Leds {
  /**
   * Turn LED n on, off, or toggle its present state.
   */
  async command void led0On();
  async command void led0Off();
  async command void led0Toggle();
  async command void led1On();
  async command void led1Off();
  async command void led1Toggle();
  async command void led2On();
  async command void led2Off();
  async command void led2Toggle();
  /**
   * Get/Set the current LED settings as a bitmask. Each bit corresponds to
   * whether an LED is on; bit 0 is LED 0, bit 1 is LED 1, etc.
   */
  async command uint8_t get();
  async command void set(uint8_t val);
}
```

**Code Sample 1: Leds Interface File**

#### 10.2.3.2 Configurations

Configurations contain a specification and an implementation. The specification specifies the interfaces that the configuration uses and provides, while the implementation contains the wiring implementation for the abstraction. The top-level application configuration for Blink, distributed with TinyOS, is presented in Code Sample 2. All wiring must be performed in configurations, where component instances that are declared as shown in Code Sample 2. Configurations can wire multiple levels of components together.

```cpp
configuration BlinkAppC {
}
implementation {
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;
  BlinkC -> MainC.Boot;
  MainC.SoftwareInit -> LedsC;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```

### Nicholas Beck

Student Number: 03971477
10.2.3.3 Modules

Modules also have a specification and an implementation. The specification contains the interfaces that the module provides and uses. However, in this case the implementation contains the real C-like implementation. The module implementation for the Blink application is presented in Code Sample 3.

```c
module BlinkC {
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface boot;
}
```

10.2.4 Concurrency Model

TinyOS has two execution threads: tasks and hardware event handlers. Tasks are non-preemptive deferred procedure calls, where components can post tasks to the scheduler for later execution. The TinyOS scheduler has a fixed-length queue that is processed in FIFO order. The implications of this were described above.

The task model is one of the largest differences between TinyOS-1.x and TinyOS-2.0 [54]. The task posting semantics are [50]:

- A task can always be posted unless it is already in the queue. The scheduler provides these semantics by using static allocation to reserve a slot in the queue for each task. This requires a byte of RAM per task (TinyOS uses two bytes per entry to store a pointer), but code can assume that task posting will never fail.

Interrupt handlers are asynchronous, and can interrupt tasks and potentially other asynchronous tasks.

10.2.5 Atomicity

Due to the run-to-completion semantics of tasks they are atomic with respect to other tasks. However, there are potential race conditions between tasks, which run in synchronous context, and interrupt handlers, which run in asynchronous context and can interrupt a task. Additionally, race conditions can occur between two asynchronous handlers [53]. In order to prevent these situations another mechanism must be employed.

Atomicity is dealt with in a very simple manner; interrupts are globally disabled and enabled. This may seem like an extreme approach, where more efficient options are available, however the highly constrained resources that are characteristic of motes require a simple solution. The thread based paradigms, provided by embedded and desktop operating systems, consume too much memory, and generate a larger processor runtime overhead.

The implications of TinyOS employing such a simple approach to atomicity, is that the developer must take responsibility of ensuring task starvation and deadlocks do not occur. However, it should be noted that atomic statements are not allowed to call commands or signal events, which confines its effects to a single module [53].

10.2.6 Resource Arbitration

There are three resource access models in TinyOS-2.x, which are described in [96]:

- Dedicated
  - Used when a subsystem needs exclusive access of a resource at all times.
  - No sharing policy is needed since only a single component ever requires use of the resource.
  - Examples of dedicated abstractions include interrupts and counter comparators.
- Shared
Shared abstractions allow clients to access the resource as if it were dedicated to them, except that they must first request the resource and wait for the granted event to be signalled.

Shared abstractions therefore require an arbiter, which is responsible for multiplexing between the different clients of a shared resource.

In TinyOS the client must explicitly release the resource. It is the developers responsibility to ensure the client does not prevent access to the resource by other clients for prolonged periods.

Bus contention is an example of where shared resource abstractions are required.

- Virtualised
  - Virtual abstractions hide multiple clients from each other through software virtualization. Every client of a virtualized resource interacts with it as if it were a dedicated resource. All virtualized instances are multiplexed on top of a single underlying resource.
  - Because the virtualization is done in software, there is no upper bound on the number of clients using the abstraction, barring memory or efficiency constraints.
  - However, virtualisation reduces the efficiency and inability to precisely control the underlying resource, as the abstraction runs in task context.
  - The system’s millisecond timer is an example of a virtualised abstraction.

### 10.2.7 Hardware Abstraction Architecture

TinyOS employs a three-tiered Hardware Abstraction Architecture (HAA) that provides both horizontal and vertical decomposition [57]. It consists of a Hardware Presentation Layer, a Hardware Adaptation Layer, and a Hardware Interface Layer.

These telescoping abstractions satisfy the requirements of both general and specialised application domains [58].

The vertical dimension spans an individual subsystem; an application to use general platform-independent interfaces, or specialised platform- and chip-specific interfaces which allow fine-grained control of the subsystems and peripherals.

Horizontal decomposition simplifies porting TinyOS to new platforms that are a combination of already supported chips. It allows reuse of subsystem implementations across different platforms.

#### 10.2.7.1 Hardware Presentation Layer (HPL)

This is the lowest level layer; it wraps the raw hardware, such as GPIO pins, interrupts, and registers, into components to provide nesC structures and interfaces for higher layers in the HAA.

#### 10.2.7.2 Hardware Adaptation Layer (HAL)

The adaptation layer components represent the core of the architecture. This layer provides APIs that enable a TinyOS application to take full advantage of the functionality provided by the hardware peripherals, whilst hiding the complexity naturally associated with the use of hardware resources; it interfaces to its respective HPL component.

A TinyOS application is able to access the HAL directly to harness all of the features of the particular hardware; however this results in the application becoming platform-specific.

#### 10.2.7.3 Hardware Interface Layer (HIL)

The HIL uses the interfaces provided by the HAL to provide platform-independent standard interfaces that can be used on all mote platforms.

The complexity of the HIL components mainly depends on how advanced the capabilities of the abstracted hardware are with respect to the platform-independent interface. When the capabilities of the hardware exceed the requirements of the HIL, the interface must reduce the functionality of the
platform, as the HIL represents the typical capabilities that all platforms should be able of. Conversely, when the underlying hardware is inferior software emulation is necessary.

10.2.8 Organisation of TinyOS-2.x
The standard directories in a TinyOS-2.x distribution are:

- **tos/system/**.
  - Core TinyOS components.
  - This directory's components are the ones necessary for TinyOS to actually run.
- **tos/interfaces/**.
  - Core TinyOS interfaces, including hardware-independent abstractions.
  - Expected to be heavily used not just by tos/system but throughout all other code.
  - tos/interfaces should only contain interfaces named in TEPs.
- **tos/types/**.
  - Header files for core TinyOS subsystems.
- **tos/platforms/**.
  - Contains code specific to mote platforms, but chip-independent.
- **tos/chips/**.
  - Contains code specific to particular chips and to chips on particular platforms.
- **tos/sensorboards/**.
  - Contains code specific to particular sensorboards, but chip- and platform-independent.
- **tos/lib/**.
  - Contains interfaces and components which extend the usefulness of TinyOS but which are not viewed as essential to its operation.
  - Libraries often contain subdirectories.
- **support/make/**.
  - Contains the TinyOS build system makefiles, rules and options. A subdirectory exists for each processor-family, which handles processor-core specific configuration.
- **support/sdk/**.
  - Contains support tools for PC applications that are to interface to a WSN, usually through a base-station node. C, Java and Python APIs are provided.
- **apps/**, **apps/demos**, **apps/tests**, **apps/tutorials**.
  - Contain applications with some division by purpose. Applications usually have a build subdirectory, which contains a directory for each platform the application has been built for. It is possible to create and use subdirectories when application complexity necessitates this.

The file naming convention for TinyOS, defined in the Coding Standards TEP [59], is presented in Table 12 below.

<table>
<thead>
<tr>
<th>TinyOS File Type</th>
<th>Example Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>Foo.nc</td>
<td>Interface definition, similar to a Java interface</td>
</tr>
<tr>
<td>Public Component</td>
<td>FooC.nc</td>
<td>Intended to be used and wired outside the package</td>
</tr>
<tr>
<td>Private Component</td>
<td>FooP.nc</td>
<td>Only used and wired within the package</td>
</tr>
<tr>
<td>Header File</td>
<td>Foo.h</td>
<td>For constants that are used by multiple nesC components</td>
</tr>
</tbody>
</table>

**Table 12: TinyOS File Naming Convention**

TinyOS-1.x used a naming convention that differentiated between modules and configurations; this has been changed in TinyOS-2.x to only distinguish between whether a component is public or private. This is more reflective of how the components should be treated by higher level components that use these; a configuration is an abstraction of its component parts, often a subsystem, where its type (configuration or module) is immaterial to the component that uses it.

10.3 Justification for Porting TinyOS-2.x
TinyOS has a steep learning curve, which is widely acknowledged [54], [60], [62], [63], and shown by the extremely high frequency of posts to the help mailing list [61].

Nicholas Beck
Student Number: 03971477
The structure of TinyOS code is rather obscure and complicated (for example, the multiple parts to an application component), and presents a steep learning curve for those unfamiliar to the language. [62]

However, TinyOS represents a very carefully thought-out approach to network implementation under conditions of limited resources, namely power and memory at each node.

TinyOS-2.0 is a clean slate redesign and re-implementation of TinyOS, with significant differences to TinyOS-1.x; an overview of these is provided in [63]. The aim of TinyOS-2.x is to address the evolving needs of WSN applications, which have been discovered through research performed using TinyOS-1.x. TinyOS-2.x offers substantial improvements over TinyOS-1.x in terms of robustness, more flexible abstractions, and energy efficiency through better resource management.

The architecture of TinyOS-2.x pushes the complexity into the HAA, which comprise the chip, platform, and sensorboard components. This has resulted in easier application development, reducing the learning curve for the vast majority of its users and increasing accessibility for researchers not from computing disciplines.

Widespread adoption of recently available second generation hardware, such as the Chipcon CC2430, has been prevented by a lack of support by WSN operating systems.

TinyOS has by far the largest user-base of any WSN OS; porting the CC2430 to it presents the most valuable contribution.

11. Porting TinyOS-2.x to the Chipcon CC2430

Upon starting this work TinyOS-2.x was a beta release and all of the TinyOS Extension Proposals (TEPs) were drafts. As a result the core components and subsystem interfaces that are expected were subject to changes. The author decided that the best course of action was to subscribe to the TinyOS-2.x CVS commits mailing list which provides notification of all source and TEP commits, including diffs of the modifications; this allows changes to be analysed so that appropriate amendments can be performed on the source code for the CC2430.

There are a number of other public TinyOS mailing lists, including tinyos-help and tinyos-devel, of which the author is an active subscriber of.

There are a large number of significant differences between TinyOS-1.x and TinyOS-2.x, which are discussed where appropriate in this section.

In the following section <TinyOS> refers to the base directory of ones TinyOS-2.x installation and <TOS> refers to the base TinyOS source directory <TINYOS>/tos/, which contains the files that implement the chips, platforms and sensorboards supported by TinyOS, plus the common interfaces, libraries and core system components.

The diagrams presented in the following section are expressed in a modified version of UML, which was necessary to express the design of subsystems and applications in TinyOS, a legend describing the aspects of the diagrams is shown in Figure 10 below.
11.1 8051 Toolchain Issues

There are known toolchain issues with porting an 8051-cored platform to TinyOS-1.x [64], [102]; these equally apply to TinyOS-2.x.

TinyOS and nesC are dependent on GCC [65], the GNU Compiler Collection, for some stages of compilation; this issue is discussed in detail in Section 11.2. However there is not a variant of GCC for the 8051 processor core. The other issues are related to the 8051’s idiosyncrasies, which require compiler specific extensions; these are described in Section 11.4.

11.1.1 Compiler Choice

Due to the reliance on GCC there is no option that will work immediately. In fact there are a number of options that could be pursued: there are a number of commercial compilers available for the 8051, an open-source 8051-capable compiler, or GCC could be ported for the 8051. These options are discussed below.

11.1.1.1 Port GCC for the 8051

The most obvious option, which would inflict less pain in the future, is to port GCC for the 8051 core. This would allow TinyOS applications to be built for any 8051-based mote. However, there are a number of factors that pertain to GCC being unsuitable for the 8051; it cannot be easily or efficiently ported.

Because the 8051 architecture is not supported by GCC, porting it for the 8051 would be a time consuming process, which is unsuitable for this project. Wilson, who works for Specifix a GNU tools porting and support specialist [66], states in a GCC mailing list post that [67]:

A quick rule of thumb is that it will take 3 months to do a port from scratch for a typical RISC target. This assumes you know what you are doing, or have someone you can ask for help. If not, it may take longer.

This correlates with information in mailing list posts by Taylor [68] and Olson [69], and obtained from a discussion with Craig Duffy. It should be noted that Olson acknowledges that “Basic Code generation can be done fairly quickly, but getting really good code generation out requires more work”. Due to the extremely limited resources on WSN devices, optimal code generation is essential.

Figure 10: Diagram Legend

<table>
<thead>
<tr>
<th>GenericComponent(type param1, type paramN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#new Component(val1, va1N)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interface Provided By Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Used By Component</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ProvidedInterface</th>
</tr>
</thead>
</table>

| ObjectInstanceOfGenericComponent: GenericComponent(type param1, type paramN) |
| new Component(val1, va1N)                                                         |

<table>
<thead>
<tr>
<th>Another Standard Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Some</td>
</tr>
<tr>
<td>+Provided</td>
</tr>
<tr>
<td>+Interfaces</td>
</tr>
<tr>
<td>+ParameterisedInterfaceUsed []</td>
</tr>
</tbody>
</table>

This correlates with information in mailing list posts by Taylor [68] and Olson [69], and obtained from a discussion with Craig Duffy. It should be noted that Olson acknowledges that “Basic Code generation can be done fairly quickly, but getting really good code generation out requires more work”. Due to the extremely limited resources on WSN devices, optimal code generation is essential.
GCC is specifically aimed at CPUs with several 32-bit general-purpose registers and byte-addressable memory [70, Ch12]. Also, gcc is not suited to segmented-memory architectures, particularly where memory is larger than can be addressed by a single register [71, Ch3]. These are all properties that the 8051 does not possess. The 8051 has 8 general purpose registers, in comparison to 16 on ARM 7 and 9 cores, and 32 on Atmel AVRs.

The 8051’s largely segmented memory (separate SFR, DATA, CODE & XDATA memory spaces exist) and limited stack operations make porting GCC an extremely challenging task [72]. Taking the enhanced 8051 core on the CC2430 as an example, the 16-bit memory space must be indirectly addressed through its 8-bit data pointer registers (Figure 11). Although it would be possible to access FLASH data using a mechanism like the PROGMEM macros [73] utilized by AVRs, which are 8-bit, and a software stack could be employed to work-around the limitations on the 8051, the compiled code would be very inefficient; this makes it unsuitable for WSNs.

Figure 11: CC2430 Memory Map [28]

11.1.1.2 Modify nesC to create code syntactically correct for other compilers

This would require creating a new build environment through the support scripts for the target compiler. Also, the nesC compiler would have to be modified to add support for the target compiler’s extensions. Currently only GCC’s extensions are supported; the nesC compiler, understandably, generates compilation errors when unknown compiler-specific language extensions are parsed. Whilst this would be possible, there are certain negative aspects to this. Compiler-specific extensions for each target compiler would be required; Table 13 shows the extent of this, every 8051 compiler has its own extensions, with a unique syntax. Further, this would encourage developers to write compiler-specific code, which is not desirable. Also, if the modifications are not integrated into the mainline distribution, new releases will potentially require further changes to the compiler patches and support scripts.

<table>
<thead>
<tr>
<th>8051 SFR declaration in IAR compiler:</th>
</tr>
</thead>
<tbody>
<tr>
<td>__sfr __no_init volatile uint8_t  P0 @ 0x80;</td>
</tr>
<tr>
<td>8051 SFR declaration in SDCC compiler:</td>
</tr>
<tr>
<td>sfr at 0x80 P0;</td>
</tr>
<tr>
<td>8051 SFR declaration in KEIL compiler:</td>
</tr>
<tr>
<td>sfr P0 = 0x80;</td>
</tr>
</tbody>
</table>

Table 13: Comparison of Register Declarations for Different 8051 Compilers

11.1.1.3 Modify the syntax created by nesC to a syntax for other compilers

This option involves creating a syntax translation script, to convert the GCC syntax output by the nesC compiler to that which is syntactically correct for the target compiler. This approach has a number of possible implementation options: the nesC support scripts can be modified to invoke the translation script, or the compilation rules of GCC can be overridden; a discussion of the various approaches is presented in Section 11.3. Additionally, the use of a syntax translation script allows idiosyncrasies to be handled; even in platform-independent components.
11.1.1.4 Conclusion

It is evident that GCC can not be ported to quickly, and that it would not generate efficient code anyway, which is of great importance to resource-constrained WSN devices.

Modifying nesC would prove possible, but is likely to result in platforms that have to use an unsupported toolchain becoming coupled with a particular compiler; currently this would result in 8051-cored platforms being coupled with a commercial compiler (Section 4). Additionally, adding support for the compiler-specific extensions does not deal with all idiosyncrasies. For example, none of the 8051 compilers discussed in Appendix A handle 64-bit integers; TinyOS core and library components which use these would not compile, as support for extensions would not transform these. The fact that this technique will only have effect on components written specifically for the platform makes it unsuitable.

The “syntax translation script” method allows all corner cases to be handled. It also makes platform code more portable to other compilers. A syntax translation script could be written that is capable of transforming from GCC syntax to that of numerous others, selected using a command-line switch for instance. Additionally, this technique has been successfully used in prior work by a number of groups, in 8051-based TinyOS-1.x ports [76 - 103], which should reduce risk.

11.1.2 Considerations for Other Architectures without GCC Support

Porting GCC for the 8051 is inappropriate, which leads the author to question whether the tight coupling of nescc and gcc (creating TinyOS’s dependency on GCC) is justifiable; especially when other low-power processor cores and future mote architectures are considered [74].

Virk et al. [75] decided to use a MCU with a different processor core than initially intended for their project; the decision was taken based solely on the initial MCU, which was a Motorola HCS08, not being supported by GCC.

A number of research groups have ported, or partially ported, platforms with MCU architectures that do not have GCC support. The platforms, shown in Table 14, are a mixture of integrated and unintegrated hardware, however all ports were targeted at TinyOS-1.x; no work has been published regarding porting TinyOS-2.x to a platform without GCC. It should be noted that all platform ports for TinyOS-2.x, in the public domain, have been performed by members of the TinyOS Core Working Group.

<table>
<thead>
<tr>
<th>Mote Platform &amp; Group</th>
<th>MCU Processor-Family &amp; Part</th>
<th>Compiler</th>
<th>Type</th>
<th>TinyOS Version Ported</th>
</tr>
</thead>
<tbody>
<tr>
<td>- / Helmut-Schmidt University &amp; EnOcean [76]</td>
<td>PIC18F / 18F452</td>
<td>Microchip C18</td>
<td>Unintegrated</td>
<td>TinyOS-1.x</td>
</tr>
<tr>
<td>- / Cork Institute of Technology [77]</td>
<td>PIC16F / 16F877</td>
<td>HITECH PICC</td>
<td>Unintegrated</td>
<td>TinyOS-1.x</td>
</tr>
<tr>
<td>WISENET / Bradley University [103]</td>
<td>8051 / CC1010</td>
<td>Keil</td>
<td>Integrated</td>
<td>TinyOS-1.x</td>
</tr>
<tr>
<td>RISE / University of California, Riverside [107]</td>
<td>8051 / CC1010</td>
<td>SDCC</td>
<td>Integrated</td>
<td>TinyOS-1.x</td>
</tr>
<tr>
<td>nRF24E1 / DIKU [102]</td>
<td>8051 / nRF24E1</td>
<td>Keil</td>
<td>Integrated</td>
<td>TinyOS-1.x</td>
</tr>
<tr>
<td>MC13192 / DIKU [78][16]</td>
<td>HCS08 / MC13192</td>
<td>Unknown, Proprietary (source is compiled remotely on machine @ DIKU)</td>
<td>Integrated</td>
<td>TinyOS-1.x</td>
</tr>
</tbody>
</table>

Table 14: Previous TinyOS Port Attempts to Platforms Without GCC Support

Appendix H discusses the need to support other compilers. Additionally, a solution is proposed to unify the mechanisms by which this is achieved. Due to space limitations one aspect of the solution was not given the discussion it deserved. The proposed solution is dependent on the target compiler, or
transformation script, having the capability of handling banked memory if it exists on the MCUs architecture. For example, the PIC16 port [77] necessitated a script for distributing variables across memory banks; this cannot be handled in platform-independent components through attribute annotation.

11.1.3 8051 Compiler Choice

The IAR 8051 toolchain was chosen for this project. For a discussion of the compiler options available and a justification for this decision see Appendix A.

11.2 TinyOS Build System

A description of the TinyOS build system, and how a new platform with an unsupported toolchain can be added is presented in Appendix B.

11.3 The Build Process

The orthodox build process is illustrated in Figure 12. It can be seen that there is a reliance on GCC for stages pre-, and post-nesC compilation. Additionally the actual nesC compiler, nesc1, only supports GCC language extensions. The other non-GNU stages are support scripts for the nesC compiler, whose actions are described in Table 15. This build process must be modified to integrate an unsupported toolchain, but it is useful to understand how it normally works.

The first stage that performs any real compilation work is “gcc (pre-processing)”, by cpp (C Pre-Processor). The pre-processed source is passed through to the nesC compiler which generates the program’s call graph, performs static analysis, and generates C code output. This C code is compiled, assembled, and linked by the GCC suite for the processor core of the target platform.

![Figure 12: Standard TinyOS Build Process](image)

<table>
<thead>
<tr>
<th>Script</th>
<th>Actions performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>ncc</td>
<td>An extension to nescc. It invokes nescc with additional options based on the platform selected. This pulls in the Perl snippets that define the directories to include in .platform and .sensor files for the specified platform and sensorboard.</td>
</tr>
<tr>
<td>nescc</td>
<td>nescc is an extension to gcc that knows how to compile nesC applications. When invoked on a nesC component or interface file it compiles and links that component with the other files specified on the command line.</td>
</tr>
<tr>
<td>nesc-compile</td>
<td>Is a wrapper around the actual nesC compiler. It separates the nesC compilation options</td>
</tr>
</tbody>
</table>

Nicholas Beck

Student Number: 03971477
from the gcc compilation options, and feeds the appropriate arguments to each compiler.

### Table 15: Actions of nesC Compiler Support Scripts

#### 11.3.1 Necessary Changes to Chip-specific Definitions

The implications of the nesC compiler only supporting GCC’s extensions mean component code cannot contain any of the target compiler’s language extensions. The nesC compiler does not know how to handle these, so will generate errors and abort compilation. The result is that chip-specific definitions, which necessitate language extensions, must be expressed in a way supported by GCC. For example, due to the highly segmented memory architecture all 8051 compilers have a language extension for defining SFRs. Further, the IAR compiler requires SFRs to be referred to by the name that has been declared. This can be likened to a variable declaration, because it is not possible to cast the address of the SFR to the correct memory segment, as can be done with XDATA; an example of this is shown in Table 16.

#### 8051 SFR declaration in IAR compiler:

```c
__sfr __no_init volatile uint8_t P0 @ 0x80;
```

**SFR must be accessed using its declared name i.e.:**

- `P0 = 0xFF;`

#### 8051 XDATA declaration in IAR compiler:

```c
#define RFPWR XREG( 0xDF17 )
```

Which is pre-processed to, and allows access to XDATA registers, using:

- `((unsigned char volatile __xdata *) 0)[0xDF17] // RFPWR radio register`

### Table 16: SFR & XDATA Memory Access Examples

The modifications to chip-specific definitions, so that they are parseable by the nesC compiler, include the SFRs and ISRs (Interrupt Service Routines). The transformations that are performed on these are described in 11.4. SFR declaration (Table 17) utilises an enum, which allows the name of the SFR to be preserved after pre-processing; this is necessary in order for the translation script to generate the required IAR syntax for the declaration, shown in Table 16. The SFR names are prepended by “SFR_” for easy detection. The result of this is that SFRs access in component implementations is performed by simply treating the SFR name as a variable.

A similar technique is performed for the ISRs, as shown in Table 18. The translation script performs a larger amount of transformation for ISR decalarations, which must be declared using IAR’s compiler-extensions (Table 19) before compilation of the generated C source can be performed.

### SFR Macro Functions

```c
#define SFR(name,addr)   SFR_##name = addr,
```

**Definition of registers:**

```c
enum sfr
{
    /* Port 0 */
    SFRBIT( P0 , 0x80, P0_7, P0_6, P0_5, P0_4, P0_3, P0_2, P0_1, P0_0 )    /* */
    SFR( SP , 0x81 )    /* Stack Pointer */
    SFR( DPL0 , 0x82 )    /* Data Pointer 0 Low Byte */
    SFR( DPH0 , 0x83 )    /* Data Pointer 0 High Byte */
    SFR( DPL1 , 0x84 )    /* Data Pointer 1 Low Byte */
    SFR( DPH1 , 0x85 )    /* Data Pointer 1 High Byte */
    SFR( U0CSR , 0x86 )    /* USART 0 Control and Status */
    SFR( PCON , 0x87 )    /* Power Mode Control */
    ...

    /* Port 1 */
    SFR( ... )
}
```

**Example SFR usage in TinyOS components:**

```c
//get the state of the LSB
P0 &= 0x01;
```

### Table 17: SFR Declaration

#### ISR Macro Functions
#define VECT(num, addr, isr_name) isr_name=addr

Definition of ISRs:

#define RFERR_VECTOR(isr_name) VECT(0, 0x03, isr_name) /* RF TX FIFO Underflow and RX FIFO Overflow */
#define ADC_VECTOR(isr_name) VECT(1, 0x0B, isr_name) /* ADC End of Conversion */
#define URX0_VECTOR(isr_name) VECT(2, 0x13, isr_name) /* USART0 RX Complete */
#define URX1_VECTOR(isr_name) VECT(3, 0x1B, isr_name) /* USART1 RX Complete */
#define ENC_VECTOR(isr_name) VECT(4, 0x23, isr_name) /* AES Encryption/Decryption Complete */
#define ST_VECTOR(isr_name) VECT(5, 0x2B, isr_name) /* Sleep Timer Compare */
#define P2INT_VECTOR(isr_name) VECT(6, 0x33, isr_name) /* Port 2 Inputs */

... 

enum isr 
{
    RFERR_VECTOR(RFERR_isr),
    ADC_VECTOR(ADC_isr),
    URX0_VECTOR(URX0_isr),
    URX1_VECTOR(URX1_isr),
    ENC_VECTOR(ENC_isr),
    ST_VECTOR(ST_isr),
    P2INT_VECTOR(P2INT_isr),
    ...
};

Example ISR declaration in TinyOS components:

NONATOMIC_HANDLER(P0INT_isr)
{
    // ISR implementation
    ...
}

Table 18: ISR Declaration

IAR-specific ISR definition:
#pragma vector=0x03
__interrupt void RFERR_isr(void);

IAR-specific ISR declaration:
__interrupt void RFERR_isr(void)
{
    // Handler implementation
    ...
}

Table 19: ISR Declaration Format Using IAR-specific Language Extensions

In hindsight, with the conscious decision to not support SFR arguments to generic components, the
definition of SFRs and ISRs could be made more elegant. By defining them as variables their name and
value would be preserved through both the pre-processor and nesC compiler, allowing them to be
transformed by the syntax translation script to the appropriate format. As they would appear to be
normal variables to the compilers, they would accept the syntax. Because arguments to generic
component must be constants, so that instances can be statically created at compilation, it was not
possible to use this technique.

11.3.2 Build Process Modification

The build process had to be modified to integrate an unsupported toolchain. Specifically, an additional
stage had to be added, a syntax translation script, and the target compiler must be invoked instead of
GCC. There are a number of possible ways that invocation of this script can be added to the nesC build
suite, and that the target compiler selection can be overridden.

The approach employed by groups in previous works has varied. For instance, T8051WG [102] leave
the main build process untouched, invoking the entire build with a script instead (Code Sample 4); this
results in the gcc compiler being invoked, which subsequently aborts, followed by the invocation of the
mangle script and target compiler. Whilst this works, it requires this script to be placed in every
application’s top directory, and gcc generates substantial errors, which mask any nesC warnings.

#!/bin/sh
make mcs51
.sdccMangleAppC.pl $1 -file build/mcs51/app.c > build/mcs51/app_mangled.c
echo ++++++++++++++++++++++++++++
keil51.exe build/mcs51/app_mangled.c -o build/mcs51/app_mangled.hex
WISENET [103] explicitly modify nesc-compile, so that it invokes the translation script and the target compiler. This requires a patch, and therefore potential modifications, to be applied for every new compiler release. It also prevents a different target compiler from being used easily for the same platform, as the two become tied. The effect on the build process of this technique is shown in Figure 13.

![Figure 13: Modified TinyOS Build Process Implemented by WISENET [103]](image1)

As both approaches have limitations an alternative solution was sought. The method used, shown in Figure 14, utilises the capabilities of gcc, which is effectively a rule-based dispatcher front-end that wraps the GCC tools. Specifically, an additional specs file is specified that overrides the default compilation rules (Code Sample 5). This approach requires a modification to the architecture makerules, such that the build is performed in two stages, highlighted in Code Sample 6. The first step only executes upto and including the nesC compiler, which generates the C source code output. The second stage overrides the default .c rule and builds the final executable.

![Figure 14: Modified TinyOS Build Process Implementation for the CC2430](image2)
Porting TinyOS-2.x to the Chipcon CC2430

Code Sample 5: Additional specs File, which Overrides Default Behaviour

```
.c:
 iarMangleAppC.pl -IAR -file build/cc2430/app.c -o build/cc2430/mangled_app.c
 icc8051.exe //command-line options
```

Code Sample 6: Modifications to the Standard Architecture Rules for the 8051 Processor

This technique allows a different target compiler for the same platform to be easily invoked, by simply specifying a different command line option. All that is required to invoke additional compilers is a .extra file and associated specs file for the target compiler; the iar.extra file is shown in Code Sample 7.

It should be noted that the process is not currently fully automated; however it is listed for future work see Section 12.1. As active development was being undertaken, and debugging requires the IAR IDE, the author did not deem this necessary.

```
exec0: builddir $(BUILD_EXTRA_DEPS) FORCE
@echo "    compiling $(COMPONENT) to a $(PLATFORM) binary"
@echo "    $(MCS51FLAGS)"
$(NCC) -v $(OPTFLAGS) $(MCS51FLAGS) $(PFLAGS) $(CFLAGS) $(WIRING_CHECK_FLAGS) $(COMPONENT).nc
@echo "    compiling $(COMPONENT) to a $(PLATFORM) binary"
@cp $(COMPONENT).c $(BUILDDIR)/app.c
$(NCC) -v -o $(MAIN_EXE) $(MCS51FLAGS) $(OPTFLAGS) $(CFLAGS) $(WIRING_CHECK_FLAGS) $(BUILDDIR)/app.c $(LIBS) $(LDFLAGS)
```

Code Sample 7: Compiler Choice through Command-line Option – iar.extra

11.4 Syntax Translation Script

The syntax translation script is written in Perl, a general purpose scripting language, which was employed for its powerful pattern matching and regular expression capabilities. The script used by T8051WG [102], for the Keil compiler and TinyOS-1.x, was used as a starting point for the basis of this project’s requirements.

A comparison of the actions taken by T8051WG [102] to those required by this project is presented in Table 20. Additional actions that are required for TinyOS-2.x and the IAR compiler are presented in Table 21.

```
#--*-Makefile-*- vim:syntax=make
MCS51FLAGS = -D__IAR_SYSTEMS_ICC__ -specs $(TINYOS_MAKE_PATH)/mcs51/iarspecs.txt
```

Table 20: Comparison of Necessary Actions by the Syntax Translation Script for the 8051 & Keil/IAR/SDCC Compiler

<table>
<thead>
<tr>
<th>Action Performed by Script [102] (Keil)</th>
<th>Required for TinyOS-2.x (&amp; nescc v1.2) with IAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFR &amp; SBIT declaration conversion</td>
<td>Conversion to IAR format instead of Keil or SDCC</td>
</tr>
<tr>
<td>Conversion of 64-bit integer types to 32-bit types</td>
<td>The IAR compiler does not support 64-bit values either</td>
</tr>
<tr>
<td>Alter instances of reserved keyword data to _data</td>
<td>data is not a reserved keyword for the IAR compiler</td>
</tr>
<tr>
<td>Removal of inline directives</td>
<td>Not required for the IAR compiler, it supports inline</td>
</tr>
<tr>
<td>Removal of pre-processor line numbers</td>
<td>Still performed.</td>
</tr>
<tr>
<td>Change ‘$’ in identifiers to ‘__’</td>
<td>Nesc provides an option to substitute the ‘$’ character</td>
</tr>
<tr>
<td>Convert GCC interrupt declaration to Keil format</td>
<td>Conversion to IAR format instead of Keil or SDCC</td>
</tr>
</tbody>
</table>

Table 21: Comparison of Necessary Actions for TinyOS-2.x and the IAR Compiler

<table>
<thead>
<tr>
<th>Action Performed by Script</th>
<th>Requirement for Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic component SFR translation</td>
<td>Arguments to generic components must be either types or constants, which requires SFRs to be cast to simple types as discussed in Section 11.8.1. The script must translate the casted SFR address back into a name for reasons discussed in Section 11.3.1</td>
</tr>
<tr>
<td>Removal of nesc_sillytask’s</td>
<td>The nesC compiler generates typedef’s which are not used. The compilation problem is caused when the typedef is to a zero-length array, which is not supported by IAR. As these are not used at all, it is sufficient to</td>
</tr>
</tbody>
</table>

Nicholas Beck

Student Number: 03971477
For struct’s with no fields, insert void; field IAR will warn that the declaration does nothing after transformation, but compilation will be successful. Originally just the struct’s braces were removed, but this does not work for nx_struct network endianness formats.

<table>
<thead>
<tr>
<th>Table 21: Additional Actions Required by the Syntax Translation Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>A second pass is performed to handle generic component SFR translation. The SFR name-address pairs are cached on the first pass, allowing the script to translate the SFR addresses (which are annotated so that they can be detected) to their names; as required by the IAR compiler. The two-pass approach means the script does not need explicit knowledge of the SFRs in order to translate them. The script was modified to create a file, rather than dumping its output to stdout, so that it did not require invocation from within a shell script.</td>
</tr>
</tbody>
</table>

### 11.5 Debugging

It is an important consideration on how debugging will be performed for an embedded device that one is porting an operating system (OS) to, immaterial of which OS.

However, debugging TinyOS is incredibly awkward due to the additional compilation level, and the obfuscation this causes; an insight into the effects of which is presented in Appendix C.

### 11.6 Sensorboards

Support for two sensorboards was created: the RF04EB, which is part of the CC2430 Development Kit, and a custom MARCO sensorboard. These are described in Appendix D and Appendix E respectively.

These appendices contain substantial work, which discusses the considerations and implications for the devices and sensors provided by these sensorboards; the designs and implementations of which are documented. Additionally, these components are the basis for subsystem unit tests, and provide examples of how the subsystems are used.

### 11.7 Bootstrap & Board Initialisation

The C runtime library (CRT) provided with the IAR compiler suite was utilised for board bring-up to main(). From here the TinyOS boot sequence [79], [80] is started, which has four stages:

- **Scheduler Initialisation.**
  - This allows component initialization routines to post tasks. The boot sequence runs tasks after each initialization stage in order to allow long-running operations, as shown in Code Sample 8, since they only happen once.

- **Component Initialisation.**
  - Platform-specific initialization is performed by a component named PlatformC, which is discussed in Section 11.7.1 below. Service specific initialisation is handled by the SoftwareInit interface, which utilises the one-to-many wiring paradigm to initialise all components that have been wired to either MainC’s or RealMainP’s SoftwareInit interface.

- **Signal boot process completion (the Boot.booted event).**
  - Applications handle the Boot.booted event, which signals the completion of the boot process. The execution of the actual application is then started. The booted event is TinyOS’s analogue of main in a Unix application.

- **Run the scheduler loop.**
  - Once the system is booted the core scheduler loop is entered, which is discussed in Section 0.
These are implemented by the core system component `RealMainP`, shown in Code Sample 8.

```c
module RealMainP {
    provides interface Booted;
    uses {
        interface Scheduler;
        interface Init as PlatformInit;
        interface Init as SoftwareInit;
    }
}
implementation {
    int main() __attribute__((C, spontaneous)) {
        atomic {
            call Scheduler.init();
            call PlatformInit.init();
            while (call Scheduler.runNextTask());
            call SoftwareInit.init();
            while (call Scheduler.runNextTask());
        }
        __nesc_enable_interrupt();
        signal Boot.booted();
        call Scheduler.taskLoop();
        return -1;
    }
    default command error_t PlatformInit.init() { return SUCCESS; }
    default command error_t SoftwareInit.init() { return SUCCESS; }
    default event void Boot.booted() { }
}
```

**Code Sample 8:** The TinyOS-2.x Boot Sequence Implementation – `RealMainP`

### 11.7.1 Platform-specific Initialisation

`PlatformC` is responsible for platform-specific initialisation; it pass-through wires its `Init` interface to `PlatformP`. The CC2430’s initialisation is shown in Code Sample 9. Most initialisation is application-specific and is therefore performed by service and device abstractions, which wire to `SoftwareInit`.

The only initialisation performed is the configuration of the clock source to use the 32MHz crystal, which the radio necessitates, and explicitly setting the global clock divisor to one, as it does not work (see Section 11.10.1 for details). Note that a spin-loop is used when waiting for the stabilisation of the oscillator, as interrupts should not be enabled before the boot sequence has completed, see TEP107 [79] for details.

```c
#include "hardware.h"

module PlatformP {
    provides interface Init;
}
implementation {
    command error_t Init.init() {
        error_t retVal;  //function return value
        /* Setup the system clock for 32MHz timer ticks
         * Note: setting to other divisor is broken on my hardware!!
         */
        /* turn on both oscillators (xtal & rc) */
        SLEEP &~ 0x04;
        /* wait for oscillator to stabilise*/
        while ({SLEEP & 0x40} == 0);
        /* switch to crystal oscillator*/
        CLKCON &~ 0x47;
        CLKCON = (_8051_CLK_DIV_1 << 3);
        return SUCCESS;
    }
}
```

**Code Sample 9:** CC2430 Platform-specific Initialisation - `PlatformC`

### 11.7.2 TinyOS Task Scheduler

The capability exists of specifying a custom task scheduler, which is discussed in “Schedulers & Tasks” TEP106 [81]. However, only the standard scheduler is discussed here.

The standard scheduler exhibits the following behaviour:
• The scheduler runs as long as there are tasks on the queue.
• When the task queue is empty the MCU is put into the lowest power state allowed by the active hardware resources.
• The processor goes to sleep until it handles an interrupt. When an interrupt arrives, the MCU exits its sleep state and runs the interrupt handler. Having handled the interrupt the task scheduler is restarted.
• If the interrupt handler posted one or more tasks, the scheduler runs tasks until the task queue and then returns to sleep.

The most important aspect of the scheduler loop, is putting the MCU into the lowest possible power state, when the queue becomes empty. This state will vary depending on which peripherals are currently active. For instance, if an alarm is set, utilizing hardware Timer3 for example, the MCU would have to remain in full-power mode as Timer3 does not operate in any other mode; it would wait for a longer period than specified as it would only count whilst the MCU was active.

As this was the first subsystem which had to be ported, and limited debug facilities existed, an implementation which did not provide power management was created; the device continuously ran at full-power. Later, after the proof-of-concept LED blinking application was tested, power management was added. This approach reduced the number of potential causes of error.

A component named `McuSleepC` is expected to provide the power management logic. The “Microcontroller Power Management” TEP [82] dictates `McuSleepC` should have the interface presented in Code Sample 10. The power management implementation for the CC2430 is shown in Code Sample 11. The state of each peripheral is checked to establish the lowest possible power state that can be selected without causing side-effects. It can be seen from Code Sample 11 that the CC2430’s low-power modes severely limit the operations that can be performed.

```plaintext
component McuSleepC {
    provides interface McuSleep;
    provides interface PowerState;
    uses interface PowerOverride;
}
```

Code Sample 10: The Dictated Interface Signature of the `McuSleepC` Component
else if (READ_BIT(U0CSR, 0) != 0)
    { //yes, must remain in full-power mode
        return _8051_PM0;
    }
else if (READ_BIT(U1CSR, 0) != 0)
    { //yes, must remain in full-power mode
        return _8051_PM0;
    }
// ADC is enabled
else if (READ_BIT(ADCCON1, 6) != 0)
    { //yes, must remain in full-power mode
        return _8051_PM0;
    }
// sleep timer enabled? Yes, always on.
else if (1) {
    return _8051_PM2;
}
}
async command void McuSleep.sleep() {
    powerState = mcombine(getPowerState(), call McuPowerOverride.lowestState());
    atomic
    {
        SLEEP &= ~(0x3);
        SLEEP |= (&i8051PowerBits[powerState]) & 0x3;
    }
    /* enable interrupts so chip can be awakened */
    __nesc_enable_interrupt();
    SET_BIT(PCON, 0); //actually put it to sleep
    __nesc_disable_interrupt();
}
async command void McuPowerState.update() {
    atomic dirty = 1;
}
default async command mcu_power_t McuPowerOverride.lowestState() {
    return _8051_PM3;
}

Code Sample 11: Power Management Functionality - McuSleepC

The CC2430’s power modes offer poor flexibility and energy conservation; the chip must operate in full-power mode if any peripheral is in use, with the exception of the sleep timer. Whilst this does not appear too awful, as most peripherals will only be accessed when they are required, a couple of things must be given thought. Some peripherals’ operations, such as the ADC, are not instantaneous but require the device to operate at full-power; if there is no other work to perform, then this wastes energy. All timers, except the sleep timer, require full-power operation. If accurate timing is required within TinyOS the chip will always run in full-power mode; this is discussed in Section 11.10 “Timers”.

McuSleepC is an example of where the current chip directory structure does not offer enough flexibility. The power modes provided by 8051-cored MCUs varies, resulting in the need for a chip-, or chip-family-specific power management implementation. This would be resolved by adopting the directory structure proposed in Appendix H.

11.8 GPIO

The author determined to develop a simple proof-of-concept application which toggles an LED, which would show whether porting TinyOS-2.x to an integrated mote platform with no supported toolchain was possible, further it would highlight whether the IAR compiler and Chipcon CC2430 were capable of this. In order to create this proof-of-concept application numerous components, which implement a small number of subsystems, would be required: boot and board initialisation, scheduler and GPIO.

It is not possible to implement an explicit set of GPIO components that are generic for all 8051-cored MCUs due to the substantial differences between hardware from different silicon vendors. For example the nRF24E1, from Nordic Semiconductor, has three ports of which one is solely dedicated for use by the radio and another has only three available pins where one is hardwired as an input. In comparison
the CC2430 has three ports that can be used as GPIO, of which one has only five pins, whilst Atmel’s AT89C51AC3 has five ports that can be used for GPIO, of which one has only five pins.

The Low-level I/O TEP [83] for TinyOS-2.x states that GPIO capabilities must be provided through a HIL layer GeneralIO interface (Code Sample 12), which provides control of an individual pin. TinyOS assumes that the pin can be configured as an input or output, and that it can be either arbitrarily sampled or set and reset, respectively. This interface should be provided for each GPIO capable pin in a component named GeneralIOC.

```c
interface GeneralIO
{
    async command void set();
    async command void clr();
    async command void toggle();
    async command bool get();
    async command bool isInput();
    async command bool isOutput();
}
```

**Code Sample 12: HIL Layer GeneralIO Interface**

The GeneralIOC component is chip-specific and resides in TinyOS’s chip directory. The initial design for the GeneralIOC component, the top-level configuration of the GPIO subsystem, used generic port components, which in turn used generic pin components, as shown in Figure 15. Whilst this appears as though it will generate an efficient implementation, using only three components which are instantiated a number of times, when the nesC source is compiled it generates highly inefficient C code. The implementation of this design also highlighted a number of issues with generic components, some of which are specific to the 8051 while others are intrinsic to TinyOS, affecting all MCU cores; these issues are discussed in Section 11.8.1.

Generic components are much like classes in high level object oriented languages, such as C++ and Java, in that object instances of the component are created, however the substantial difference is that object instantiation of nesC generic components is performed at the time of compilation, the objects are static in order for the nesC compiler to generate a call graph and perform static analysis of the program as described in Section 10.2.2. The nesC compiler performs a similar operation for generic components as the C++ compiler does with STL, where a duplicate of the component’s code and variables is created for each instance of the component, only the object’s parameters are used to replace certain unique data, such as the SFR register address for the particular port the object realises as shown in Code Sample 13.

The use of generic components results in a full, but unique, copy of the component for each instance, producing a large code footprint for the subsystem. It should be noted that if a single GPIO pin was wired to (used) by an application, the nesC compiler would only produce a single instance, and therefore copy, of the component rather than generate a copy for all possible pins on the device; however, it is likely that an application would utilise more than a single GPIO pin.

```c
static /*__inline*/   void
/*Hpl8051GeneralIOPinP__16__GeneralIO__set(void)
{
P1 |= 1 << 3;
}
```

```c
static /*__inline*/   void
/*Hpl8051GeneralIOPinP__13__GeneralIO__set(void)
{
P1 |= 1 << 0;
}
```

**Code Sample 13: Output of Generic Components Generated by the nesC Compiler: from the ButtonToLed Application**

The behaviour of the nesC compiler was discovered through inspection of the generated C code. It is suboptimal, and considered by the author to be unacceptable considering the constrained resources available on the chip. Additionally, this implementation’s top-level configuration is not easily portable to other 8051-based MCUs with a different number of, and pin configuration, for its ports. Therefore an alternative GPIO subsystem design was deemed necessary.

The alternative design is shown in Figure 16, which employs parameterised interfaces. The top-level configuration, GeneralIOC, provides a HIL layer GeneralIO interface for each GPIO pin on the
CC2430 through three parameterised port components which enables the GPIO pins to be accessed at the HIL level as highlighted in Code Sample 14. The three modules that implement the functionality for each port are identical, bar acting on the specific IO and DDR (Data Direction Register) SFRs for the respective port.

A conscious decision to not use generic components whose parameters contains SFRs was made due to the issues described in Section 11.8.1, however ideally the GPIO subsystem would be implemented as a generic module which would not only ease maintenance, but more importantly it would allow the GPIO subsystem to be used for all 8051-cored MCUs irrespective of the number of ports they provide. This would be achieved through the use of a chip-specific version of the GeneralIOC configuration that instantiates and wires to the required ports on the particular chip as shown in Figure 17.

```plaintext
configuration PlatformLedsC {
  provides interface GeneralIO as Led0;
  provides interface GeneralIO as Led1;
  provides interface GeneralIO as Led2;
  uses interface Init;
}
implementation {
  components GeneralIOC as IO;
  components PlatformP;
  Init = PlatformP.MoteInit;
  Led0 = IO.Port1[0];  // Port1 Pin1 = Green LED
  Led1 = IO.Port1[3];  // Port1 Pin3 = Yellow LED
  Led2 = IO.Port0[0];  // Port0 Pin0 = Yellow LED
}
```

Code Sample 14: Example of Wiring to the GPIO Subsystem from Higher Level Components: `PlatformLedsC`
Figure 15: Original GPIO Subsystem Design
Figure 16: Revised GPIO Subsystem Design

Figure 17: Idealised Design for GPIO on the Chipcon C2430
11.8.1 Issues with Generic Components

The generic component semantics, enforced by the nesC compiler, stipulate that all arguments for generic component parameters must be types or, numerical or character constants [56, Ch6]. The effect this has on 8051-cored platforms specifically, and the platform-independent consequences of this, are discussed below. A solution is proposed in Appendix H.

11.8.1.1 Platform-independent Issues

In order for a SFR to be passed as a parameter to a generic component a hack is required; this affects all platforms. For example, the work-around employed for the Atmega128's generic GPIO components is shown in Code Sample 15 and Code Sample 16. The hack is used to cast the generic component's constant parameters into AVR formatted SFRs. Note that the generic module shown in Code Sample 16 is instantiated by a generic configuration for the mote’s ports.

```
components new HplAtm128GeneralIOPortP((uint8_t)&PORTA,
            (uint8_t)&DDRA,
            (uint8_t)&PINA) as PortA;
```

**Code Sample 15: Atmega128 GPIO Generic Component Instantiation [84]**

```
generic module HplAtm128GeneralIOPinP (  
            uint8_t port_addr,
            uint8_t ddr_addr,
            uint8_t pin_addr,
            uint8_t bit)
          {
            provides interface GeneralIO as IO;
          }
implementation
  
#define pin (*(volatile uint8_t *)pin_addr)
#define port (*(volatile uint8_t *)port_addr)
#define ddr (*(volatile uint8_t *)ddr_addr)
  . . .
```

**Code Sample 16: Atmega128 GPIO Generic Component Hack [85]**

11.8.1.2 8051 Specific Issues

As it is not possible to declare SFRs with #defines they must be declared as variables, in order to convey both the name and address to the translation script. The result of this is that it prevents SFRs from being used in generic components. Therefore a hack is required to allow SFRs to be used as arguments to generic components. The hack involves the redefinition of all SFRs with #define and the effective annotation of each reference to a SFR with (*volatile sfr *)port_addr): this allows the syntax translation script to locate instances which require modification. The script is employed to convert the numeric address for the SFR to its name, which is required by the IAR compiler because of the language extension it uses.

Whilst this proved it is possible to work-around the issue generic component issue, the solution is not at all elegant and offers little gain. The code generated by the nesC compiler is effectively the same as writing a separate component for each instance of the generic. The intended advantage of easier code maintenance does not compensate for the additional complexity introduced. This has led the author to purposely not employ generic components that would require SFR arguments.

11.8.2 LEDs

There are no LEDs on the CC2430 evaluation module, however TinyOS assumes that every platform has at least three LEDs, which are expected to be exposed through a component named PlatformLedsC. A decision was made to treat the sensorboard’s LEDs as if they were part of the CC2430 platform. This prevents the developer from having to specify a sensorboard to make when it is immaterial which board is in use.

The design of the full LEDs abstraction, including platform-independent components provided by the library, is shown in Figure 18. The platform-specific LED components are ultimately wired to PlatformP’s Init interface. This enables the GPIO pins to be configured appropriately on system startup.
Whilst both of the sensorboards, that support was provided for in this body of work (Appendix D and Appendix F), have their LEDs connected to the same GPIO pins, this will not necessarily always be the case. The wiring of PlatformLedsC’s GeneralIO interfaces could be deferred to the application, where the selection for the sensorboard in question, or a non-functional dummy LEDs platform component, could be made. However, this would require the developer to understand that unlike all other platforms, they must perform platform-specific wiring for the LEDs. This justifies the decision to treat the LEDs as part of the platform, though future applications and sensorboards may necessitate this change.

11.8.3 Testing the Subsystem

A simple application was created to test the GPIO subsystem’s behaviour was correct. The application simply toggled an LED in a busy loop. This proved both the GPIO subsystem and the components associated with the boot sequence discussed in Section 11.7.

11.9 GPIO Interrupts

There are a number of requirements and assumptions made for GPIO interrupts, which are defined in the TEP117: Low-Level I/O [83]. The subsystem’s configuration component must provide the GpioInterrupt HIL interface (Code Sample 17), and it is assumed that, when the pin is configured as an input, the ability to generate an interrupt from either a rising edge or falling edge exists. The TEP does not state any HIL components for GPIO interrupts that TinyOS assumes exist.

```c
interface GpioInterrupt
{
    async command error_t enableRisingEdge();
    async command bool isRisingEdgeEnabled();
    async command error_t enableFallingEdge();
    async command bool isFallingEdgeEnabled();
    async command error_t disable();
}
```
TinyOS provides separate interfaces for GPIO and GPIO interrupts. Whilst it would be possible to create a component that presents both `GeneralI/O` and `GpioInterrupt` interfaces, as in Figure 19 below, this implementation would not be optimal, for the reasons discussed below.

A design decision was made to treat GPIO and GPIO interrupts as two distinct subsystems. This offers some benefits over bundling the functionality into a single subsystem. If an application does not require an interrupt request for a pin, or it is an output, then there is no need for the operations associated with the `GpioInterrupt` interface, and code size can thus be reduced.

This is actually not as clear cut, because parameterised interfaces are used which only ultimately generate a single function for each operation per port, rather than per pin. However, if an entire port has no requirement for receiving interrupt requests from GPIO pins, the approach taken of creating separate subsystems means that the ISR, and all code reachable from the ISR, will not be removed by the dead-code analysis performed by the nesC compiler. The result is a smaller code size when this functionality is not required.

The design of the GPIO interrupt subsystem is shown in Figure 20. It consists of a top-level configuration and three modules that implement the interrupt interface at a port level; which is where the ISR is required because of the different functionality provided by each port. Like the GPIO subsystem the interfaces provided are parameterised, which generates a single function per port for each operation provided by the interface. So in the common case of an application employing multiple pins on a port the generated code size is no greater than when only a single pin is used; this is an aspect the system architect should understand when considering the pin usage of an application.

`GpioInterrupt`C is a configuration that provides a HIL level abstraction to all pins of the CC2430 in the GPIO interrupt subsystem. The interfaces presented by the configuration are wired to the modules which implement each port, as shown in Code Sample 18 below, the pass-through wiring allows the pin used to be deferred to higher level components. The nesC compiler will optimise the pass-through wiring so that a method invocation on the configuration’s interfaces will actually call the method on the component on the right-hand-side (e.g. `GpioInterruptC.Port0[1]` is translated to call `Hpl8051GpioPort0P.GpioInterrupt[1]`).

The configuration wires the HIL level interfaces it provides directly to the HPL modules that implement the functionality, bypassing the HAL layer which normally exists. There are a number of reasons for bypassing the HAL:

- Due to the separate control interface that is provided, because this affects both the GPIO and GPIO interrupt systems, there is no need for the HPL modules to provide a different interface than `GpioInterrupt`.

**Figure 19: Single Configuration for GPIO & GPIO Interrupts**

A design decision was made to treat GPIO and GPIO interrupts as two distinct subsystems. This offers some benefits over bundling the functionality into a single subsystem. If an application does not require an interrupt request for a pin, or it is an output, then there is no need for the operations associated with the `GpioInterrupt` interface, and code size can thus be reduced.

This is actually not as clear cut, because parameterised interfaces are used which only ultimately generate a single function for each operation per port, rather than per pin. However, if an entire port has no requirement for receiving interrupt requests from GPIO pins, the approach taken of creating separate subsystems means that the ISR, and all code reachable from the ISR, will not be removed by the dead-code analysis performed by the nesC compiler. The result is a smaller code size when this functionality is not required.

The design of the GPIO interrupt subsystem is shown in Figure 20. It consists of a top-level configuration and three modules that implement the interrupt interface at a port level; which is where the ISR is required because of the different functionality provided by each port. Like the GPIO subsystem the interfaces provided are parameterised, which generates a single function per port for each operation provided by the interface. So in the common case of an application employing multiple pins on a port the generated code size is no greater than when only a single pin is used; this is an aspect the system architect should understand when considering the pin usage of an application.

`GpioInterrupt`C is a configuration that provides a HIL level abstraction to all pins of the CC2430 in the GPIO interrupt subsystem. The interfaces presented by the configuration are wired to the modules which implement each port, as shown in Code Sample 18 below, the pass-through wiring allows the pin used to be deferred to higher level components. The nesC compiler will optimise the pass-through wiring so that a method invocation on the configuration’s interfaces will actually call the method on the component on the right-hand-side (e.g. `GpioInterruptC.Port0[1]` is translated to call `Hpl8051GpioPort0P.GpioInterrupt[1]`).

The configuration wires the HIL level interfaces it provides directly to the HPL modules that implement the functionality, bypassing the HAL layer which normally exists. There are a number of reasons for bypassing the HAL:

- Due to the separate control interface that is provided, because this affects both the GPIO and GPIO interrupt systems, there is no need for the HPL modules to provide a different interface than `GpioInterrupt`.

### Code Sample 17: GpioInterrupt HIL Interface

```c
async event void fired();
```

TinyOS provides separate interfaces for GPIO and GPIO interrupts. Whilst it would be possible to create a component that presents both `GeneralI/O` and `GpioInterrupt` interfaces, as in Figure 19 below, this implementation would not be optimal, for the reasons discussed below.
- The HPL components provide the interface expected at the HIL layer, meaning there is no translation required between the HPL, which wraps access to the hardware, and the subsystem's top-level configuration. Inserting HAL components would unnecessarily increase the code size.

- Because the per-pin interrupts for port 0 and 2 are emulated in software they require persistent state variables. Introducing HAL layer components that perform this emulation would require either the relocation of the port's ISR to the HAL layer, which is not the correct place for it, or a set of operations for updating the HPL's software emulated interrupt mask state. This would result in a large number of calls between the HAL and HPL components reducing the runtime performance and decreasing energy efficiency.

![Figure 20: GPIO Interrupt Design](image)

```plaintext
configuration GpioInterruptC
{
    provides
    |
        interface GpioInterrupt as Port0:uint8_t pin;
        interface GpioInterrupt as Port1:uint8_t pin;
        interface GpioInterrupt as Port2:uint8_t pin;
    }
}
implementation
{
    components Hpl8051GpioPort0P;
    components Hpl8051GpioPort1P;
    components Hpl8051GpioPort2P;
    Port0 = Hpl8051GpioPort0P;
    Port1 = Hpl8051GpioPort1P;
    Port2 = Hpl8051GpioPort2P;
}
```

**Code Sample 18: Top-level Configuration of the GPIO Interrupt Subsystem**

The `GpioInterrupt` interface provides baseline event control for a GPIO pin, it does not cater for chip specific functionality. This is dealt with separately by a chip-specific control interface and module, `Hpl8051GpioPortConfig` and `Hpl8051GpioPortConfigC` respectively, which present the CC2430’s additional GPIO features, such as setting the pin as pullup/pulldown/tristate and configuring the pin for GPIO or its peripheral function.

Separating the chip-specific control functionality helps mask the differences between different MCUs with 8051 processor cores. A different control component (an HPL/HAL component that is accessed directly by higher-level components) is required for each 8051-cored MCU. Currently the directory structure of TinyOS is not suited to chips that have the same processor core, but have different implementations. However, the structure proposed in Appendix H allows reuse of code components across chip families and SoC devices. The structure provides the flexibility for core-, family-, and device-specific functional block implementations.
11.9.1 GPIO Interrupt Architecture on the CC2430

The interrupts associated with the GPIO pins cannot be dealt with using a generic configuration as each port provides different functionality and support. This presents an issue with respect to reuse of components across other 8051-cored MCUs, which may have a different number of ports and different port capabilities than the CC2430; there is no method to represent GPIO interrupts in a chip-independent way, so each 8051-cored platform needs to define its own GPIO interrupt components. The directory structure proposed by the author in [74] / Appendix H tackles the issue of having different implementations of a peripheral for different MCUs with the same processor core, organising chips by processor-core, chip-family and specific device which couples maximum component reuse and flexibility; additional features or more energy efficient implementations can be capitalised on by effectively overloading the component through its wiring configuration.

The GPIO interrupt capabilities of the CC2430 can be seen in Figure 22 below. Port1 has fully maskable interrupt capabilities for each pin, whilst Port0 provides hardware support for masking the top and bottom nibble of the port. Port2 consists of only 5 GPIO pins, which are treated as a single maskable unit in hardware.
### PICTL (0x8C) — Port Interrupt Control

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Reset</th>
<th>R/W</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>—</td>
<td>0</td>
<td>RO</td>
<td>Not used</td>
</tr>
<tr>
<td>6</td>
<td>PDASC</td>
<td>0</td>
<td>RW</td>
<td>Drive strength control for I/O pins in output mode. Selects output drive capability to account for low I/O supply voltage on pin DASC.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 Minimum drive capability, IOVDD equal or greater than 2.6V</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 Maximum drive capability, IOVDD less than 2.6V</td>
</tr>
<tr>
<td>5</td>
<td>P1IEN</td>
<td>0</td>
<td>RW</td>
<td>Port 2, inputs 4 to 0 interrupt enable. This bit enables interrupt requests for the port 2 inputs 4 to 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 Interrupts are disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 Interrupts are enabled</td>
</tr>
<tr>
<td>4</td>
<td>P0IENS</td>
<td>0</td>
<td>RW</td>
<td>Port 0, inputs 7 to 6 interrupt enable. This bit enables interrupt requests for the port 0 inputs 7 to 6.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 Interrupts are disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 Interrupts are enabled</td>
</tr>
<tr>
<td>3</td>
<td>P0IENL</td>
<td>0</td>
<td>RW</td>
<td>Port 0, inputs 3 to 2 interrupt enable. This bit enables interrupt requests for the port 0 inputs 3 to 2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 Interrupts are disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 Interrupts are enabled</td>
</tr>
<tr>
<td>2</td>
<td>P1ICOM</td>
<td>0</td>
<td>RW</td>
<td>Port 1, inputs 4 to 0 interrupt configuration. This bit selects the interrupt request condition for all port 0 inputs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 Rising edge on input gives interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 Falling edge on input gives interrupt</td>
</tr>
<tr>
<td>1</td>
<td>P1ICOM</td>
<td>0</td>
<td>RW</td>
<td>Port 1, inputs 7 to 6 interrupt configuration. This bit selects the interrupt request condition for all port 1 inputs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 Rising edge on input gives interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 Falling edge on input gives interrupt</td>
</tr>
<tr>
<td>0</td>
<td>P0ICOM</td>
<td>0</td>
<td>RW</td>
<td>Port 0, inputs 7 to 6 interrupt configuration. This bit selects the interrupt request condition for all port 0 inputs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 Rising edge on input gives interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 Falling edge on input gives interrupt</td>
</tr>
</tbody>
</table>

### P1IEN (0x8D) — Port 1 Interrupt Mask

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Reset</th>
<th>R/W</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>P1[7:0]EN</td>
<td>0x00</td>
<td>RW</td>
<td>Port P1[7] to P1[0] interrupt enable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 Interrupts are disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 Interrupts are enabled</td>
</tr>
</tbody>
</table>

Figure 22: Interrupt Enable Capabilities of the CC2430 GPIO Port

### P0IFG (0x89) — Port 0 Interrupt Status Flag

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Reset</th>
<th>R/W</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:6</td>
<td>P0IF[7:0]</td>
<td>0x00</td>
<td>R/W</td>
<td>Port 0, inputs 7 to 0 interrupt status flags. When an input port pin has an interrupt request pending, the corresponding flag bit will be set.</td>
</tr>
</tbody>
</table>

### P1IFG (0x8A) — Port 1 Interrupt Status Flag

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Reset</th>
<th>R/W</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:6</td>
<td>P1IF[7:0]</td>
<td>0x00</td>
<td>R/W</td>
<td>Port 1, inputs 7 to 0 interrupt status flags. When an input port pin has an interrupt request pending, the corresponding flag bit will be set.</td>
</tr>
</tbody>
</table>

### P2IFG (0x8B) — Port 2 Interrupt Status Flag

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Reset</th>
<th>R/W</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:5</td>
<td>—</td>
<td>00</td>
<td>RO</td>
<td>Not used</td>
</tr>
<tr>
<td>4:0</td>
<td>P2IF[4:0]</td>
<td>0x00</td>
<td>R/W</td>
<td>Port 2, inputs 4 to 0 interrupt status flags. When an input port pin has an interrupt request pending, the corresponding flag bit will be set.</td>
</tr>
</tbody>
</table>

Figure 23: GPIO Interrupt Flags on the CC2430

Where hardware maskable interrupt support is not provided on a per-pin basis this functionality must be emulated in software, as TinyOS expects an interrupt request to be signalled from each individual pin.

Nicholas Beck  
Student Number: 03971477
This is possible due to the interrupt flags that are provided for each pin of every port, as shown in Figure 23, a member variable with component scope is effectively used as a pseudo interrupt mask for the port.

The architecture of the CC2430’s GPIO interrupts limits the selection of which edge an interrupt will be generated on to a whole port. This limitation does not match the assumption made by TinyOS, that each pin’s interrupt generation edge can be individually set. From a survey of other MCUs, which have low-power processor cores and are suitable for use as a WSN platform, it is the author’s opinion that the assumption made by TinyOS of expecting edge-triggers to be configurable on a per-pin basis is reasonable.

The device is still perfectly useable in the vast majority of application scenarios, but consideration must be given to the system’s architecture at an early stage of application development to ensure that all pins configured for GPIO that generate an interrupt on a particular port use the same edge trigger. For example one port may be used for all connected devices/sensors that require GPIO interrupts to be signalled on a rising edge, and another port used for all connected devices/sensors that require GPIO interrupts to be signalled on a falling edge.

There are issues when a device connected to a pin requires interrupt requests to be generated on both the rising and falling edge; this is effectively level-triggered which the CC2430 does not have explicit hardware support for. Applications that have pins which generate an interrupt on both rising and falling edges can be implemented on the CC2430 provided one of the following techniques can be utilised:

- The device or sensor is connected to a pin, on a port that it has dedicated use of, thereby allowing the edge trigger to modified from rising to falling to rising, for example, without interfering with the behaviour of other device or sensors connected to pins on a different port. The proviso is that the device or sensor connected to the pin does not generate interrupts, that must be caught, more quickly than the pin’s edge trigger can be switched.

- The edge detection can be emulated in software, using mechanisms such as polling the pin status, or far more likely timer delayed polling. The disadvantage of this is that the timer spends more time executing, rather than being in an ultra-low power sleep, which has an energy consumption expense.
  - Timer polling has a lower energy consumption than busy loop polling as the MCU can be put into a sleep mode and only awakened when the timer interrupt occurs to check the state of the pin, however this cycle may be performed multiple times before the interrupt condition occurs, whereas the MCU can sleep until the actual event occurs with a real interrupt. This mechanism suffers from the detriment of response time to the actual event occurring.
  - Polling the pin allows the interrupt condition to be detected as soon as it occurs, but prevents execution of other tasks and could cause starvation if the interrupt condition is not almost immediate. When this is combined with the higher energy consumption, as the MCU never sleeps whilst polling in this manner, it can be seen that its uses are very rare.

- In scenarios where the device or sensor, requiring interrupt signals on both edges, runs only for short periods and the other pins that generate interrupts on the same particular port can be ignored during this time, it is possible to use a shared port.

11.9.2 Testing the Subsystem

The button abstraction, whose design and implementation is presented in Appendix D, was used to test that rising and falling edge interrupts could be configured and triggered; the button provided an easy mechanism for ensuring the GPIO interrupt subsystem exhibited the correct behaviour, as it will fire a falling-edge interrupt when the button is depressed and a rising-edge interrupt on release (as shown in Figure 24).
Because the functionality, and therefore the implementation, of each port is different, individual testing must be performed. However, there are a limited number of user input devices on the RF04EB sensorboard which prevents testing all pins like the push button, but the CC2430’s port pins are exposed on the RF04EB by two pin-row headers which allows a high, or low, logic level voltage to be applied and interrupt generation tested. For this purpose a test application was created which toggles a LED when an interrupt is generated. Further, this technique was used to test that only the correct pin’s interrupt request was serviced where a software emulated interrupt mask is used. A combination of the interactive debugger and toggling GPIO pins at runtime were used to ensure the correct functionality of the subsystem. Although the interactive debugger allows a breakpoint to be set on the ISR, and then single step through it enabling the behaviour to be checked for correctness easily, a runtime test was also performed due to the consequential effects of the interactive debugger discussed in 11.5; this was accomplished using a multimeter and an oscilloscope.

### 11.10 Timers

There are a number of timers/counters on the CC2430, which provide numerous precisions, prescalers and functionality, described in Table 22.

Timers have changed dramatically since TinyOS-1.x. The timer subsystem is incredibly more complex; it is described in TEP102 [86]. Platform and chip level implementation and component composition is far more challenging than for TinyOS-1.x (for an insight into this compare the timer source for TinyOS-1.x [87] and TinyOS-2.x [88], [89]), but this has resulted in a number of benefits:

- Using timers in applications is easier.
- Provided interfaces are more uniform (both platform-specific and platform-independent).
- Greater flexibility in use of underlying hardware timers.
- A lowest common denominator approach to the interfaces is no longer taken.
  - This tackles the issue that the 8051 Working Group (8051wg) experienced in their attempt to port an 8051-cored platform to TinyOS-1.x [102], where it was necessary to modify core interfaces.
- Tight compile-time timer interface compatibility checking.

<table>
<thead>
<tr>
<th>Timer</th>
<th>Size</th>
<th>Prescaler Divisions (Effective rate)</th>
<th>Number of Comparators</th>
<th>Set-able Counter Value</th>
<th>Overflow Interrupt</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer1</td>
<td>16-bit</td>
<td>1 (32MHz), 8 (4MHz), 32 (1MHz), 128 (250kHz)</td>
<td>3</td>
<td>Only reset to zero</td>
<td>Yes</td>
<td>Free-running, modulo, up/down modes. Only useable in full-power mode.</td>
</tr>
<tr>
<td>Sleep Timer</td>
<td>24-bit</td>
<td>N/A (32.768kHz)</td>
<td>1</td>
<td>No</td>
<td>No, not even an overflow flag.</td>
<td>Timer not suspendable. Free-running from POR (Power-On Reset). Tick derived from 32kHz oscillator. It is the only timer which can operate in any of the sleep modes.</td>
</tr>
<tr>
<td>Timer3</td>
<td>8-bit</td>
<td>1 (32MHz), 2 (16MHz), 4 (8MHz), 8 (4MHz), 16 (2MHz), 32 (1MHz), 64</td>
<td>2</td>
<td>Only reset to zero</td>
<td>Yes</td>
<td>Free-running, down, modulo, up/down modes. Numerous compare actions.</td>
</tr>
</tbody>
</table>
Table 22: CC2430 Clock Facilities

11.10.1 Hardware Issues
The global clock prescaler has no effect, resulting in a clock speed reduction derived solely from the
prescaler for the particular timer. Despite no mention in the chip’s datasheet, a potential cause of this
is the requirement of a settle period after a change to the global prescaler; an extended delay through the
use of a busy-wait loop, and the order of clock configuration were explored to no avail. This is believed
to be an issue with the silicon, as all software possibilities have been exhausted. The implications of this
are discussed below.

11.10.2 Timer Interfaces
All of the platform-independent timer interfaces are parameterised, as shown in Code Sample 19,
allowing common use by all timers. Parameterized interfaces provide the same functionality, and
perform the same actions, as the Standard Template Library (STL) does for C++. Additionally, this
intentionally makes timer interfaces with different precision or width mutually incompatible, and clearly
expresses the attributes of a given timer interface. The precision_tag parameter indicates the interfaces
tick speed (Code Sample 20) and the size_type represents the timer’s width expressed as a data type.

| Timer4 | 8-bit | (500kHz), 128 (250kHz) | Only reset to zero | Yes | Free-running, down, modulo, up/down modes. | Numerous compare actions. | Only useable in full-power mode. |

Code Sample 19: Parameterised Timer Interfaces

```c
typedef struct {} TMilli; // 1024 ticks per second
typedef struct {} T32khz; // 32768 ticks per second
typedef struct {} TMicro; // 1048576 ticks per second
```

Code Sample 20: Example Timer Precision Definitions

```c
interface Counter< precision_tag, size_type >
interface Alarm< precision_tag, size_type >
interface BusyWait< precision_tag, size_type >
interface LocalTime< precision_tag >
interface Timer< precision_tag >
```

Similarly, the chip-specific interfaces provided by HPL timer components are parameterised (Figure
25), allowing their reuse on all of the platform’s timers.

![Figure 25: Common Interfaces Used By All Timers](image-url)
11.10.3 Timer Transformations

A set of library components allow timer transformations to be performed. This offers the ability to effectively lower the timer’s precision, increase its width, or both, through software emulation.

This enables a timer to provide interfaces which match that expected by higher-level components. For example, the millisecond timer expects a timer interface with millisecond precision and a 32-bit width, which none of the underlying hardware timers can natively provide.

The generic Counter interface transformation module is shown in Code Sample 21; it contains the software emulation logic providing precision and timer widths conversions. By instantiating a TransformCounterC component with the appropriate parameters another interface, with the desired properties is created. The same hardware timer is used by the native and transformed interfaces. A component that instantiates a transformation object must wire the interfaces. An example of the counter transformation interface for the sleep timer is presented in Code Sample 22.

```cpp
generic module TransformCounterC(
    typedef to_precision_tag,
    typedef to_size_type @integer(),
    typedef from_precision_tag,
    typedef from_size_type @integer(),
    uint8_t bit_shift_right,
    typedef upper_count_type @integer() )
{
    provides interface Counter<to_precision_tag,to_size_type> as Counter;
    uses interface Counter<from_precision_tag,from_size_type> as CounterFrom;
}
```

**Code Sample 21: TransformCounterC’s Component Interface Signature**

```cpp
class configuration CounterMilli32C
{
    provides interface Counter<TMilli, uint32_t>;
}
class implementation
{
    components new TransformCounterC(TMilli, uint32_t, //Transform to
    T32khz, uint32_t, //transform from
    5, uint32_t) as Transform,
    Counter32khz32C;
    Counter = Transform;
    Transform.CounterFrom -> Counter32khz32C;
}
```

**Code Sample 22: Example Usage of TransformCounterC**

The Alarm transformation module is very similar to the counter; its interface is shown in Code Sample 23. It is a generic module allowing any number of transformations to be performed. TransformAlarmC requires an already widened Counter component, generated using the TransformCounterC component above. An example of the alarm transformation interface for the sleep timer is presented in Code Sample 24.

```cpp
generic module TransformAlarmC(
    typedef to_precision_tag,
    typedef to_size_type @integer(),
    typedef from_precision_tag,
    typedef from_size_type @integer(),
    uint8_t bit_shift_right )
{
    provides interface Alarm<to_precision_tag,to_size_type> as Alarm;
    uses interface Counter<from_precision_tag,from_size_type> as AlarmFrom;
}
```

**Code Sample 23: TransformAlarmC’s Component Interface Signature**

```cpp
generic configuration AlarmMilli32C()
{
    provides interface Alarm<TMilli, uint32_t>;
}
class implementation
```
components new TransformAlarmC(TMili, uint32_t, /*Transform to*/
T32khz, uint32_t, /*transform from*/
5) as Transform,
    new Alarm32khz32C(),
    CounterMilli32C;
Alarm = Transform;
Transform.Counter -> CounterMilli32C;
Transform.AlarmFrom -> Alarm32khz32C;

Code Sample 24: Example Usage of TransformAlarmC

The bit-shift right that changes the precision can be seen in the examples above, which effectively halves the clock speed with each binary shift. In this particular case a five bit-shifts are required to transform the 32kHz clock-rate to 1kHz. Due to the lack of a functioning global prescaler the minimum achievable clock-rate for any timer, other than the sleep timer, is 250kHz. When this is transformed to a 32kHz timer for example, through three right bit-shifts, the actual clock-rate is 31.25kHz; which amounts to a substantial difference over a period.

11.10.4 Sleep Timer

The timer TEP [86] states that all platforms must provide a HilTimerMilliC HIL component. What it does not state is that the timer must continue to count when in the MCU is in a low-power mode. This information is hidden in [90]:

*The T2 timer subsystem is built over the 8-bit timer 0, as it is the only timer that can run when the ATmega128 is in its low-power mode.*

The only timer on the CC2430 capable of running in any power mode is the sleep timer, as commented in Table 22. Therefore it must be used as the underlying hardware timer for the virtualised millisecond timer. Additionally, due to the non-functional global prescaler, other timers would require an undesirably large timer precision emulation in software (in Section 11.10.3 above).

The sleep timer’s HAL components must provide Alarm and Counter interfaces, which should be provided by components with naming convention shown in Code Sample 25 [86]. These should express the native precision and width of the timer. However this is not possible as the sleep timer is 24-bits wide, and there is no 24-bit data type. The design that realises platform-specific access to the sleep timer is presented in Figure 26.

The sleep timer is not suited to the timer subsystem architecture that TinyOS provides and expects.

Where \( P \) is the interface’s precision, and \( W \) is the interface’s width.

Counters:
configuration Counter$P$W

Alarms:
generic configuration Alarm$P$W

Code Sample 25: Suggested naming Convention of Platform-specific Counter & Alarm Components
Figure 26: Millisecond Timer Design – Lower
11.10.4.1 Millisecond Timer

The millisecond timer is the only platform-independent timer in TinyOS; all other timers provide only HAL level interfaces. Virtually every platform-independent TinyOS application uses the virtualised millisecond timer abstraction.

The millisecond timer component, \texttt{HilTimerMilliC}, expects a timer interface with millisecond precision and a 32-bit width. This poses some serious issues due to the limited functionality of the sleep timer:

- No overflow interrupt.
  - The timer overflow condition \textbf{must} be detected. This is necessary to deal with alarm expirations where the timer count will wrap.
  - Requires software emulation.
- Single comparator.
  - If a minimum of two comparators existed, one could effectively be utilised for timer overflow.
- The comparator value cannot be read (see Figure 27).
  - A shadow copy must be kept so that the comparator value can be obtained from the \textit{Alarm} interface.
- The sleep timer is 24-bits wide, but there is no 24-bit data type, so label it as a 32-bit timer. However, this has issues too.
  - A precision transformation cannot be performed because there is no overflow interrupt. This event is required by the counter transformation component for software emulation.
  - A width transformation cannot be performed because it appears to be the same; caused by the macro function employed, shown in Code Sample 26.
- Treat the alarm as a 16-bit timer, and transform it to a 32-bit timer.
  - This would introduce greater jitter, through the software emulation.
  - However, this is not possible because the timer count cannot be reset to zero. If a 16-bit comparator were set it would not be fired for potentially a full timer wrap period later than required; because the counter would have to overflow its full 24-bit count before the comparators matched.
Figure 27: CC2430 Sleep Timer Registers [28]

Code Sample 26: Timer-width Manipulation Macros

The solution was to create a pseudo native 32-bit sleep timer, through software emulation overflow interrupts which allowed software emulation of a 32-bit timer width. A member function of the component, `InternalOverflow()` shown in Code Sample 27, realises the software emulation of timer overflow interrupts. It is called by all operations that are effected by an overflow event before performing their own actions, thereby ensuring detection of the overflow.

```c
enum
|
| LOW_SHIFT_RIGHT = bit_shift_right,
| HIGH_SHIFT_LEFT = 8*sizeof(from_size_type) - LOW_SHIFT_RIGHT,
| NUM_UPPER_BITS = 8*sizeof(to_size_type) - 8*sizeof(from_size_type) + bit_shift_right,
// 1. hack to remove warning when NUM_UPPER_BITS == 8*sizeof(upper_count_type)
// 2. still provide warning if NUM_UPPER_BITS > 8*sizeof(upper_count_type)
// 3. and allow for the strange case of NUM_UPPER_BITS == 0
| OVERFLOW_MASK = NUM_UPPER_BITS ? ((((upper_count_type)2) << (NUM_UPPER_BITS-1)) - 1) : 0,
|;
```

Code Sample 27: Component Member Function `InternalOverflow()` – Hpl8051SleepTimerP

As there is not a timer overflow flag, the event is detected by checking if the previous timer count is in the future, with respect to the timer’s current count, highlighted in Code Sample 28. A timer overflow occurs approximately every nine minutes, therefore if a check for an overflow condition is performed less frequently than this the timer would miss overflow events.
inline async command uint32_t Timer.get()
{
    uint32_t timeNow;
    atomic
    {
        /* Emulate overflow interrupts in software by checking everytime */
        /* this function is called. */
        timeNow = ST0;
        timeNow |= (uint32_t)((uint32_t)ST1 << 8);
        timeNow |= (uint32_t)((uint32_t)ST2 << 16);

        if (lastTime > timeNow)
        {
            /* time has wrapped so overflow has occurred */
            InternalOverflow();
        } 
    }
    return ((upperTime) | timeNow);
}

Code Sample 28: Timer.get() Operation, Detects Timer Overflow - Hpl8051SleepTimerP

Two methods are employed to combat the possibility of missing a timer overflow. When the
comparator interrupt is disabled the comparator is set to the sleep timer’s overflow count, shown in
Code Sample 29 ensuring the overflow will be caught before the timer count wraps. The method which is
more likely to be triggered, ensures that overflows are detected when periods between method
invocations are greater than the duration between timer overflows. This is handled by firing a compare
interrupt at least once every timer period, but only signalling the event to client components when it
corresponds to the 32-bit compare value they have specified, as shown in Code Sample 30.

/* disable the compare interrupt */
inline async command void Compare.stop()
{
    atomic
    {
        comparatorEnabled = FALSE;
        CLR_BIT(IEN0, 5);
    }
    call Compare.set(0xFFFFFF);
}

Code Sample 29: Catch Timer Overflow Events, Trap Method 1 - Hpl8051SleepTimerP

inline async command void Compare.set(uint32_t t)
{
    atomic
    {
        upperOverflow = (t >> 24);
        /* check if a emulated overflow time has been specified */
        if (upperOverflow != 0)
        {
            upperOverflowInUse = TRUE;
        }
        comparatorValue = (t & 0xFFFFFF);
        ST2 = comparatorValue >> 16;
        ST1 = comparatorValue >> 8;
        ST0 = comparatorValue;
    }
}

.. .

/* ISR for all Sleep Timer interrupt (prototype & implementation) */
NONATOMIC_HANDLER(ST_isr)
{
    if (comparatorEnabled == TRUE)
    {
        /* is it a pseudo overflow (i.e. requires a 32-bit data type) */
        if (upperOverflowInUse == TRUE)
        {
            /* just decrement the emulated upper byte count */
            /* upperOverflowInUse --;
            if (upperOverflow == 0)
            { /* emulated upper byte of 32-bits is complete */
                upperOverflowInUse = FALSE;
            }
        }
    }
}
else
{
    signal Compare.fired();
}
else
{ /* the comparator is not actively being used, but its set to */
    /* something, so this must signify genuine 24-bit overflow */
    InternalOverflow();
}
IRCON &= ~(0x80);

Code Sample 30: Catch Timer Overflow Events, Trap Method 2 - Hpl8051SleepTimerP

The virtualised millisecond timer implementation uses the service instance design pattern, shown in Figure 28 [60]. As the state of all virtualised timers is known the implementation can easily determine which timer has to fire next, it can then schedule the underlying clock resource to fire as few interrupts as possible to meet the lowest timer’s requirement. Firing fewer interrupts reduces CPU load on the system and can allow it to sleep longer, saving energy.

A virtualised millisecond timer is ultimately created from the component composition shown in Figure 29; this design which provides the platform-independent millisecond timer connects to the platform-specific design shown in Figure 26. In theory this allows an infinite number of millisecond precision timers to be instantiated from a single underlying hardware timer; in reality a limit of 255 is imposed by the uint8_t data type of the parameterised interface (highlighted in Code Sample 33), which should be more than sufficient for virtually any WSN application. It should be noted that although the millisecond timer design looks over-engineered, the nesC compiler optimises the cross-component calls so that a direct method invocations of the target components interfaces are performed.
Virtualisation of the timer is achieved using the two compile-time constant functions which are evaluated at compilation [54]. An application creates an instance of TimerMilliC, shown in Code Sample 31, for each timer it requires. This utilises unique(), which generates a unique numerical identifier for each instance with the specified name. TimerMilliP (Code Sample 32) performs the wiring conversion of the single Timer Interface, requested by TimerMilliC, to the parameterised Timer interface provided by HilTimerMilliC (Code Sample 33). HilTimerMilliC creates the virtualized timers, whose logic is implemented by VirtualizeTimerC. This component contains the state of all virtualized timers; its parameter specifies the number of which so they can be statically allocated. The uniqueCount() constant function evaluates at compilations to the number of instances generated with unique().

```
generic configuration TimerMilliC() {  
  provides interface Timer<TMilli>;  
} 
implementation {  
  components TimerMilliP;  
  // The key to unique is based off of TimerMilliC because TimerMilliImplP  
  // is just a pass-through to the underlying HIL component (TimerMilli).  
  Timer = TimerMilliP.TimerMilli[unique(UQ_TIMER_MILLI)];  
}

#include "Timer.h"

class configuration TimerMilliP {  
  provides interface Timer<TMilli> as TimerMilli[uint8_t id];  
} 
implementation {  
  components HilTimerMilliC, MainC;  
  MainC.SoftwareInit -> HilTimerMilliC;  
  TimerMilli = HilTimerMilliC;  
}

class configuration HilTimerMilliC {  
  provides interface Init;  
  provides interface Timer<TMilli> as TimerMilli[uint8_t num];  
} 
implementation {  
  enum {  
    TIMER_COUNT = uniqueCount(UQ_TIMER_MILLI)  
  };  
  components new AlarmMilli32C(),  
    new VirtualizeTimerC(TMilli, TIMER_COUNT),  
    new AlarmToTimerC(TMilli),  
    Init32khzP;  
  Init = Init32khzP;  
  TimerMilli = VirtualizeTimerC;  
  VirtualizeTimerC.TimerFrom -> AlarmToTimerC;  
  AlarmToTimerC.Alarm -> AlarmMilli32C;  
}
```

Code Sample 31: TimerMilliC - Top-level Configuration that Creates a Virtual Millisecond Timer

Code Sample 32: TimerMilliP – Private Configuration that Wires the Single Timer Interface Requested by TimerMilliC to the Parameterised Timer[] Interface Provided by HilTimerMilliC

Code Sample 33: HilTimerMilliC - HIL Component that Provides the Virtualised Millisecond Timer

The virtualised timer executes in a synchronous task context. This introduces jitter from two sources; there is an undeterminable delay between the specified expiration count and when the task is scheduled to deal with this, and when multiple timers expire simultaneously they are handled sequentially. If accurate, timely, responses to alarm expiration are required the platform-specific Alarm<precision><width>C and Counter<precision><width>C HAL components must be used.
11.10.4.2 Implications
The CC2430’s sleep timer is the only timer capable of operating in a reduced power mode. Combined with its limited capabilities severely hampers the device’s suitability as a platform for WSN applications. Software emulation reduces its performance, but the largest impact is caused by its single comparator and TinyOS’s requirement for a millisecond timer [86].

The single comparator is dedicated to the virtualised millisecond timer, which executes in task context as described above. Therefore if an application requires accurate timing it must employ another timer, resulting in the chip running at full-power.

It would be possible to use the sleep timer’s platform-specific Alarm and Counter interfaces directly, preventing the use of the millisecond timer. However, this would only provide a single component with timely acknowledgements of its specified period expiring; any other component requiring a timer would be forced to utilise a different underlying hardware timer, and the full-power mode this necessitates.

11.10.4.3 Testing the Virtualized Millisecond Timer & Underlying Sleep Timer
Testing was performed using the platform-independent Blink application [91] distributed with TinyOS, which creates three millisecond timer instances which have different periods. The period of each of the timers was checked using an oscilloscope, which was connected to the header that exposes the CC2430’s pins. A soak test was performed to ensure the correct behaviour of the overflow emulation.

11.10.5 Timer1 & BusyWaitMicroC
The Timer TEP [86] states that a platform must have a BusyWaitMicroC component, which has the signature shown in Code Sample 34. Hardware Timer1 was chosen for the implementation of this component, because it is ideally suited to the required interface; it is 16-bits wide and can be prescaled to 1MHz.

```
configuration BusyWaitMicroC
{
  provides interface BusyWait<TMicro,uint16_t>;
}
```

Code Sample 34: Component Interface Signature of BusyWaitMicroC
The design of Timer1, shown in Figure 30, only ultimately provides a BusyWait interface. There is no reason why Alarm & Counter components could not be created for this timer, which could be in turn transformed. However, due to the abundance of timers the CC2430 is equipped with there has been no requirement for this. If they were to be created they would be very similar to the alarm and counter components used by the other timers.
Figure 30: Design of BusyWaitMicroC, Realised Through Timer1

The implementation of BusyWait’s wait() operation, highlighted in Code Sample 35, is a spin-loop with support for handling timer overflow. The timer’s count, which increments every microsecond, is retrieved and compared to the desired expiration count.

generic module i8051BusyWaitC(typedef frequency_tag, typedef timer_size @integer())
    [ provides interface BusyWait<frequency_tag,timer_size> as BusyWait;
    uses interface Hpl8051Timer<timer_size> as Timer;
    ]
implementation
    [ inline async command void BusyWait.wait(timer_size duration)
    { timer_size now;
      timer_size endtime;
      now = call Timer.get();
      endtime = now + duration;
      /* check for timer wrap */
      if (endtime < now)
      { /* busy wait until the timer overflow occurs */
        while (endtime < (call Timer.get()))
        { /* busy wait until specified duration has ended */
          /* Perform the busy wait part where no overflow occurs
             This is the correct action where no overflow occurs, but is required
             for overflow to finish the wait between 0 & the endtime. */
          while (endtime > (call Timer.get()))
          { /* busy wait until specified duration has ended */
            /* overflow event does not affect the wait */
        }
      }
    }
}
Code Sample 35: BusyWait Interface Implementation

11.10.5.1 Testing BusyWaitMicroC & Timer1

An application was created with a set busy-wait duration, which toggled a GPIO pin on completion of the wait; this was repeatedly performed. The busy-wait duration was observed with an oscilloscope to ensure it performed accurately.

11.10.6 Timer3 & Timer4

Timer3 and Timer4 have identical behaviour. All references in this section are to Timer3 in the interests of brevity; however this can be read as Timer3 or Timer4.

Timer3 must provide Alarm and Counter interfaces, which should be provided by HAL components with the naming convention shown in Code Sample 25 [86]. These should express the native precision and width of the timer.

Where \([P]\) is the interface's precision, and \([W]\) is the interface's width.

Counters:

configuration Counter\(\{P\}\{W\}\)C

Alarms:

generic configuration Alarm\(\{P\}\{W\}\)C

Code Sample 36: Suggested naming Convention of Platform-specific Counter & Alarm Components

The design for Timer3, which provides Alarm and Counter interfaces through platform-specific HAL components Alarm250khz8C and Counter250khz8C, is presented in Figure 31.

![Figure 31: Timer3 Design; Presents Alarm & Counter Interfaces with Native Size & Precision](image)

Alarms must be generic components, because they each require a dedicated comparator. Compile-time checking of comparator over-use is enforced using the mechanism shown in Code Sample 37 and Code
Sample 38, where the important aspects are highlighted. This utilises the *dispatcher* pattern, discussed in [60] and illustrated in Figure 32. The timer’s HAL-level alarm component, Alarm250khz8C, wires the alarm instance’s compare interface to the parameterised interface provided by Hpl8051Timer3C (Code Sample 37). It uses `unique()`, whose unique numerical identifier will be zero for the first instance, and automatically increments for each new instance. Hpl8051Timer3C explicitly wires its parameterised compare interface to specific compare interfaces in Hpl8051Timer3P (Code Sample 38). This causes a compilation error to be generated if too many alarm instances are created, as they cannot be wired in Hpl8051Timer3C. Additionally, Hpl8051Timer3C’s parameterized interfaces will only include code for the alarms generated, resulting in a smaller code footprint.

![Figure 32: The Dispatcher Design Pattern](image)

The timer transformation components, discussed in Section 11.10.3 above, are utilised to create emulated 32kHz, 32-bit Alarm and Counter interfaces; these are transformed from the hardware timer’s native interfaces, which present its true precision and width. The component composition used to achieve this is shown in Figure 33.
Figure 33: Timer3 Transformations to provide 32-bit, 32kHz Alarm & Counter Interfaces

11.11 SMBus / I²C

SMBus and I²C [92] are essentially compatible protocols (see [93] for significant differences). They provide a relatively low-speed, low-power, two-wire bus that allows communication with a diverse range of devices and sensors.

The CC2430 does not have an I²C/SMBus controller; therefore a bit-banged implementation was required, whose design is presented in Figure 35. SMBus data transfer is achieved as shown in Figure 34; full details are available in the SMBus Specification [94].

Figure 34: Data Transfers on SMBus [94]

It utilises generic components so that any number of pseudo controllers, and therefore buses, can be created using two GPIO pins each. Pass-through wiring is used to defer wiring to the client abstraction or application, as they have knowledge of the pin assignments for the clock and data line of the device in question. Code Sample 39 and Code Sample 40 provide examples of how this is achieved; the deferred wiring of the SMBus lines to GPIO pins is highlighted in Code Sample 40.
using this design, refer to Appendix D.

Code Sample 39: SoftwareSmbusC Configuration

configuration LcdC
{
   provides interface Lcd;
}

implementation
{
   components LcdP, MainC, GeneralIOC, new SoftwareSmbusC() as SMBus;

   Lcd = LcdP;
   MainC.SoftwareInit -> LcdP.Init;
   LcdP.Bus -> SMBus;
   /**< Wire the clock & data lines to the appropriate GPIO pins */
   SMBus.ClkLine -> GeneralIOC.Port2[0];
   SMBus.DataLine -> GeneralIOC.Port1[2];
}

Code Sample 40: LCD Using the Bit-banged SMBus Implementation – LcdC Configuration

No resource arbitration is provided by these components to minimize the code size if this functionality is not required; specifically, when there is only a single SMBus device wired to a pseudo controller instance. For a discussion of how resource arbitration should be achieved, and considerations when using this design, refer to Appendix D.
This subsystem was tested using the LCD panel on the RF04EB sensorboard. The LCD’s design and implementation are presented in Appendix D.

A bit-bang implementation is more expensive with respect to the amount of computation required for a data transfer, and therefore energy consumption as the MCU is active for a longer period. This is because the MCU’s firmware must service the pseudo-controller every time the clock line toggles (every bit transferred, plus start/stop), so that data can be read from or written to the slave devices on the bus. In comparison a dedicated hardware I^2C controller, usually, fires a single interrupt on completion of a byte, or packet, transmission or reception; allowing the processor to spend more time in a low power sleep mode.

A dedicated hardware I^2C controller is an unfortunate omission from the CC2430 considering:

- The number of I^2C sensors that are available.
- That GPIO pins are expensive in terms of the chips physical footprint (one of the largest constraints on the chip’s size) and energy expenditure.
- That multiple devices can be attached to, and individually addressed on, a single two-wire bus.

### 11.12 Analogue-to-Digital Converter ADC

The ADC functionality provided by MCUs varies dramatically, as shown in Table 23, which prevents an orthodox HIL layer. The ADC subsystem obeys the three-tier HAA [57] paradigm, though it only uses a "weak" HIL, described in TEP101 [95]; the configuration, usage and implementation of the ADC is dependent on the capabilities of the hardware in question. The HIL-layer interfaces, that the ADC subsystem provides, effectively perform a callback into the sensors platform-specific HAL-layer component, to obtain the ADC configuration required (Figure 36).

<table>
<thead>
<tr>
<th>Resolution</th>
<th>CC2430</th>
<th>Atmel Atmega 128</th>
<th>TI MSP430 ADC12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channels</td>
<td>8 multiplexed individual input channels</td>
<td>8 multiplexed external channels</td>
<td>8 individually configurable external channels</td>
</tr>
<tr>
<td></td>
<td>4 differential voltage combinations</td>
<td>16 differential voltage input combinations</td>
<td>Internal channels (avcc, temperature, reference voltages)</td>
</tr>
<tr>
<td></td>
<td>Internal channels (avdd, temperature)</td>
<td>2 differential inputs with gain amplification</td>
<td></td>
</tr>
<tr>
<td>Internal reference</td>
<td>1.25V</td>
<td>2.56V</td>
<td>1.5V or 2.5V</td>
</tr>
<tr>
<td>Voltage</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conversion reference</td>
<td>Reference voltage selectable as: internal, external single-ended, external differential or AVDD_SOC.</td>
<td>Positive terminal: avcc or 2.56V or AREF (external) Negative terminal: GND</td>
<td>Individually selectable per channel: Avcc and avss Vref+ and avss Vref+ and avss Avcc and (vref- or veref-) Vref+ and (vref- or veref-) Vref+ and (vref- or veref-)</td>
</tr>
<tr>
<td>Conversion modes</td>
<td>Single channel conversion, using “extra” channel Sequence mode: 1 to 8 (single-ended channels), 9 to 12 (differential inputs), or single conversions for 13 to 15. See [28] for details</td>
<td>Single channel conversion mode Free running mode (channels and reference voltages can be switched between samples)</td>
<td>Single conversion mode Repeat single conversion mode Sequence mode (sequence &lt;= 16 channels) Repeat sequence mode</td>
</tr>
<tr>
<td>Conversion clock source</td>
<td>32MHz system clock divided by 8; giving 4MHz effective</td>
<td>Clkadc with prescaler</td>
<td>ACLK, MCLK, SMCLK or ADC-oscillator (5mhz) with</td>
</tr>
</tbody>
</table>
The ADC TEP [95] states that a platform must have AdcReadClientC, AdcReadNowClientC, and AdcReadStreamClientC components, which provide Read, ReadNow, and ReadStream interfaces respectively; the HIL interfaces listed above are shown in Code Sample 41.

```
interface Read<val_t>
{
  command error_t read();
  event void readDone( error_t result, val_t val );
}

interface ReadNow<val_t>
{
  async command error_t read();
  async event void readDone( error_t result, val_t val );
}

interface ReadStream<val_t>
{
  command error_t postBuffer( val_t* buf, uint16_t count );
  command error_t read( uint32_t usPeriod );
  event void bufferDone( error_t result,
                         val_t* buf, uint16_t count );
  event void readDone( error_t result );
}
```

**Code Sample 41: ADC HIL Interfaces**
The design of the ADC subsystem is shown in Figure 37; for clarity the only HIL interface shown is `Read`. The `ReadNow` interface is similar to the `Read` interface, but works in an asynchronous context. It does not handle resource requests like `Read`, but pass-through wires the `Resource` interface, delegating responsibility for resource acquisition to the client. The `ReadStream` interface is effectively the same as `Read`, only with the addition of a microsecond alarm, whose expiration starts another conversion.

`AdcReadClientC` instantiates both a resource arbiter and a resource handler. This removes all associated complexity with, and potential misuse of, shared resources from client components.

Figure 37: Design of the ADC Subsystem

Use of the ADC is limited to single conversions because of its interrupt triggers. The ADC only generates an interrupt on completion of an "extra" conversion. Sequence conversions do not generate any interrupts whatsoever. The "extra" channel must be set for each sample required as the hardware automatically clears it.

Whilst a DMA channel can be used to effectively create another interrupt, as stated in Table 23, its trigger condition is not particularly beneficial. For a DMA interrupt request to be signalled requires sequential channel conversion to be performed, additionally there is an overhead associated with DMA channel setup. The configuration of these is limited to an end channel, so potentially every channel must
be sampled, and transferred by the DMA, before the required interrupt is generated. The MCU must remain in full-power mode for the duration of this, resulting in high energy consumption.

To support the ReadStream interface, which is mandatory at the HIL level, requires software emulation. The CC2430’s ADC cannot generate an interrupt for each conversion when sampling a channel repeatedly, as the “extra” channel must be explicitly set each time. Therefore the HIL layer component implementation performs multiple single conversions; the interval between conversions is specified by the client component.

The CC2430’s ADC cannot be used in low-power modes. However, considering sequential conversions are not really useable, and the high conversion speed at lower resolutions (Table 24), this is of little consequence; particularly when the wake-up time from low-power sleep modes, 89.2 µs, is considered.

<table>
<thead>
<tr>
<th>ADC Sample Resolution (Decimation rate)</th>
<th>Duration: $T_{conv} = (\text{decimation rate} + 16) \times 0.25 \mu s$ [28]</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit (64)</td>
<td>20 µs</td>
</tr>
<tr>
<td>10-bit (128)</td>
<td>36 µs</td>
</tr>
<tr>
<td>12-bit (256)</td>
<td>68 µs</td>
</tr>
<tr>
<td>14-bit (512)</td>
<td>132 µs</td>
</tr>
</tbody>
</table>

Table 24: ADC Conversion Times

11.12.1 Testing the ADC Subsystem
ADC subsystem testing is described in Appendix D.

11.12.2 Integrated Temperature Sensor
According to the documentation the CC2430 is equipped with an integrated temperature sensor, which is connected to the ADC [28], [121]. However it appears to be non-functional or non-existent, which corresponds with Chipcon’s action of removing it from their demonstration application.

11.13 UART
The layers of the full UART subsystem are shown in Figure 38; the top three levels are platform-independent, which means that from the perspective of porting the UART for TinyOS only the layer highlighted in grey must be implemented, which must conform to the interfaces and behaviour expected by the encoder/framer layer above it.

![Figure 38: The Full UART Subsystem](image)

The UART subsystem uses two platform-independent generic components, SerialAMSenderC and SerialAMReceiverC, which connect to SerialActiveMessageC to provide virtualized access to the serial stack [96]. Each instance has its own single depth queue preventing it from contending with other instances for queue space, and masks the fact that the CC2430’s UARTs only have a single byte buffer for each data direction. The underlying implementation provides a fair-queuing scheduler. For further details see TEP113 [97], it should be noted that at the time of writing the SerialByteComm interface described in this document does not correlate with the source tree.
The UART interfaces are shown in Figure 39. The HPL-level control interface contains SPI operations as the controller can be configured to provide UART or SPI functionality. However, there was no requirement for SPI, nor was it possible to test, so it has not been implemented in this body of work; SPI is generally used when high-speed, high-bandwidth, inter-chip data transfers for communication are required, such as the radio on most mote platforms, but the CC2430’s is integrated.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hpl8051UsartControl</td>
<td>-enableUARTMode() : void</td>
</tr>
<tr>
<td></td>
<td>+enableSpiMode() : void</td>
</tr>
<tr>
<td></td>
<td>+enableTxIrq() : void</td>
</tr>
<tr>
<td></td>
<td>+enableRxIrq() : void</td>
</tr>
<tr>
<td></td>
<td>+disableTxIrq() : void</td>
</tr>
<tr>
<td></td>
<td>+disableReceiver() : void</td>
</tr>
<tr>
<td></td>
<td>+enableFlowControl() : void</td>
</tr>
<tr>
<td></td>
<td>+disableFlowControl() : void</td>
</tr>
<tr>
<td></td>
<td>+enableParity() : void</td>
</tr>
<tr>
<td></td>
<td>+disableParity() : void</td>
</tr>
<tr>
<td></td>
<td>+setNumStopBits(numStopBits : enum 8051NumStopBits_e) : error_t</td>
</tr>
<tr>
<td></td>
<td>+setNumDataBits(numDataBits : enum 8051NumDataBits_e) : error_t</td>
</tr>
<tr>
<td></td>
<td>+setBitOrder(bitOrder : enum 8051BitOrder_e) : error_t</td>
</tr>
<tr>
<td></td>
<td>+setSpiMode0(mode : enum 8051SpiMode_e) : error_t</td>
</tr>
<tr>
<td></td>
<td>+setSpiClockPolarity(polarity : enum 8051SpiClockPolarity_e) : error_t</td>
</tr>
<tr>
<td></td>
<td>+setSpiClockPhase(0 : enum 8051SpiClockPhase_e) : error_t</td>
</tr>
<tr>
<td></td>
<td>+setBaud(baud : uint8_t, in baud_exp : uint8_t) : error_t</td>
</tr>
<tr>
<td></td>
<td>+flush() : void</td>
</tr>
<tr>
<td></td>
<td>+setFramingError() : bool</td>
</tr>
<tr>
<td></td>
<td>+setParityError() : bool</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hpl8051Data</td>
<td>+isRxEmpty() : bool</td>
</tr>
<tr>
<td></td>
<td>+isTxEmpty() : bool</td>
</tr>
<tr>
<td></td>
<td>+isBusy() : bool</td>
</tr>
<tr>
<td></td>
<td>+done(in data : uint8_t) : void</td>
</tr>
<tr>
<td></td>
<td>+done(in data : uint8_t) : void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UartStream</td>
<td>+disableReceiveInterrupt() : error_t</td>
</tr>
<tr>
<td></td>
<td>+enableReceiveInterrupt() : error_t</td>
</tr>
<tr>
<td></td>
<td>+receive() : error_t</td>
</tr>
<tr>
<td></td>
<td>+send() : error_t</td>
</tr>
<tr>
<td></td>
<td>+receiveByte() : uint8_t</td>
</tr>
<tr>
<td></td>
<td>+receiveDone() : uint8_t</td>
</tr>
<tr>
<td></td>
<td>+sendDone() : uint8_t</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UartByte</td>
<td>+receiveByte(in byte_ptr : uint8_t, in timeout : uint8_t) : error_t</td>
</tr>
<tr>
<td></td>
<td>+send(in byte : uint8_t) : error_t</td>
</tr>
</tbody>
</table>

Figure 39: HPL, HAL & HIL Level UART Interfaces

The CC2430 has two UART controllers; one is presented as the platform’s serial port. A HIL-level configuration named PlatformSerialC is expected to be found in the platform directory, which the platform independent components wire to, as described above. The platform’s serial port is configured to use UART0 as shown in Figure 41. The HAL component, i8051UsartC, is responsible for wiring the instance of the generic module i8051UsartP to the appropriate interfaces of the HPL component.

If a component used i8051Usart1C, it would create and wire another instance of i8051UsartP to UART1’s interfaces provided by Hpl8051UsartP.

A generic module, i8051UsartP, contains the state machine for data communications (Figure 40). It is an interrupt-driven device driver, which loads the next byte in the stream on receipt of an interrupt indicating the UART’s transmit buffer is free (see the snippets in Code Sample 42). This realises a split-phase implementation [54] by maintaining some component state. Because the transmission of data is not instantaneous a polled approach would starve queued tasks. It should be noted that during the Tx’ing Byte state in Figure 40 any queued tasks will be executed or the MCU will sleep, until another byte can be transmitted.
/* Variables at component scope */
uint8_t *rxBuf = NULL;
uint8_t *txBuf = NULL;
uint16_t rxBufLength;
uint16_t txBufLength;
uint16_t rxBufPos;
uint16_t txBufPos;
...

/* new stream being transmitted, initialise internal data variables */
async command error_t UartStream.send( uint8_t *buf, uint16_t len)
{
    if (len == 0)
        return FAIL;
    atomic
    {
        //check the tx buffer is not in use
        if (txBuf)
            return EBUSY;
        txBuf = buf;
        txBufLength = len;
        txBufPos = 0;
        call Data.tx(txBuf[txBufPos++]);
        return SUCCESS;
    }
}
...
async event void Data.txDone()
{
    if (txBufPos < txBufLength)
    {
        call Data.tx(txBuf[txBufPos++]);
    }
    else
    {
        uint8_t *buf;
        buf = txBuf;
        txBuf = NULL;
        signal UartStream.sendDone(buf, txBufLength, SUCCESS);
    }
}

Figure 40: UART Transmission State Machine

Code Sample 42: UART Data Transmission State Machine - i8051UsartP.nc

Hpl8051P is another instance where parameterised generic components would reduce code duplication (see Section 11.8.1 for the reasoning behind explicitly not using generic components), as the functionality provided by each controller is identical.
The UART controller does not function correctly. It fails to transmit data if a short delay is not introduced between bytes, despite being double buffered and the hardware generating an interrupt signalling it is ready to accept another byte. The datasheet states [28]:

An interrupt request is generated when the UxDBUF register is ready to accept new transmit data.

The solution was discovered from an observation of the different behaviour exhibited between debug and release executables. The debug executable transmitted incorrect data, whilst the release build transmitted nothing. One of the two differences between the builds, executable size and execution speed, were likely to be the cause of the problem. Considering the fully static nature of TinyOS applications the executable size was unlikely to be the cause, as different behaviour from this is generally caused by pointer misuse; therefore timing was investigated.

A delay between byte transmission is achieved using a busy-wait provided by BusyWaitMicroC. A busy-wait is applicable as the necessary period is small; putting the chip into low-power mode momentarily before awakening it would cause a greater delay for no real gain in energy efficiency. As a result Hpl8051UsartP uses an additional interface which is wired to BusyWaitMicroC by i8051Usart0C, shown in Figure 42. The two UARTs are able to share the BusyWait interface as its operations are called from an atomic context\(^1\), preventing concurrent execution.

---

\(^1\) It should be noted that its operations are called from an atomic context, making them atomic, rather than the BusyWait interface’s operations being atomic themselves. However, it was essential that the busy-wait period in this scenario is minimal, as concurrency is prevented during this time.
11.13.1 Printf()

Recently a printf() style debug aid was added to TinyOS-2.x, which is provided by PrintfC in TinyOS’s library of platform-independent reusable components. This uses traditional printf syntax, with the exception that flushing must be undertaken through the use of an interface.

Whilst it is acknowledged that this will affect both the timing and executable size of the application it provides a simple and rapid method of providing feedback to the developer on the state of an executing application. If this facility had existed previously the implementation of the UART subsystem would have been promoted to one of the first subsystems implemented.

11.13.2 Devices Connected to Motes

It should be noted that if a device is connected directly to a mote, such as a GPS receiver, it must be wired directly to the HIL-layer PlatformSerialC or HAL-layer i8051Uart0; the reason for this is the platform-independent layers. The transmitter adds, and the receiver expects, additional packet formatting. If the format is considered incorrect the packet is discarded.

The implication of wiring directly to the HIL or HAL is that resource arbitration no longer exists, however it is highly likely that a device connected directly to the mote will only be accessed by a single client. This is true for all TinyOS platforms, but provided the implications are known and considered (which they are not explicitly stated) it was a fully reasonable design decision by the core working group.

11.13.3 Testing the UART Subsystem

A platform-independent test application, TestSerial, is distributed with TinyOS. It consists of a Java application that runs on a PC, and a TinyOS application that runs on the mote. Both applications transmit and receive a payload that contains a count, which is incremented for every transmission. The PC application displays the received count from the attached mote in the console, whilst the mote displays the three least significant bits of the received count on its LEDs. A soak test was successfully performed, proving the reliability of the UART subsystem.

11.14 Radio

The Chipcon CC2420 is the radio transceiver used for RF communication duties on the Chipcon CC2430. This radio is also available as a discrete IC, which is used in a number of motes that have TinyOS support (such as the MicaZ, Imote2.0, and Telos). Due to the separation of platform-agnostic chip code from platform-dependent platform code in TinyOS-2.x the implementation of the CC2420 is platform-independent. However, the CC2420 implementation is not connection agnostic; the discrete radio IC is connected to the MCU via SPI as shown in Figure 43 below.
The author determined to reuse the existing CC2420 implementation so any bug-fixes or performance enhancements can be instantly taken advantage of. This is especially important considering it has been stated on the TinyOS development mailing list [99], that the standard implementation is likely to be changed for a low-power listening mechanism in the short future. The upper levels of the radio abstractions are described in TEP126 [100].

A number of differences exist between the integrated and standalone versions of the radio, which are presented in Table 25.

<table>
<thead>
<tr>
<th>Property</th>
<th>Integrated CC2420 on CC2430</th>
<th>Standalone CC2420 as Discrete IC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection type</td>
<td>Integrated</td>
<td>SPI bus</td>
</tr>
<tr>
<td>Chip Select</td>
<td>No</td>
<td>Yes of course, its an SPI bus connected device</td>
</tr>
<tr>
<td>Radio registers</td>
<td>Either 1 or 2 eight bit radio registers in XDATA</td>
<td>16-bit, accessed in a single operation</td>
</tr>
<tr>
<td>Radio RAM (Lengths vary)</td>
<td>Most accessible (IEEE Address, PAN ID, Short Address). Accessed through multiple registers, the number of which varies with the size of “RAM” being read.</td>
<td>All accessible (IEEE Address, PAN ID, Short Address, Tx FIFO RAM). All bytes for segment read in a single operation.</td>
</tr>
<tr>
<td>Strobe commands</td>
<td>Performed through RFST register. Ability to run instructions on the radio co-processor, through another set of instructions. Equivalent functionality to the standalone version is provided, but the strobe values are different.</td>
<td>Performed over SPI. Only equivalent to “immediate instructions” provided by the SoC version.</td>
</tr>
<tr>
<td>Addresses</td>
<td>Radio register and RAM addresses are different to standalone version</td>
<td>Radio register and RAM addresses are different to standalone version</td>
</tr>
<tr>
<td>Status Byte</td>
<td>Equivalent status byte to standalone version requires harvesting bits from numerous radio registers</td>
<td>Returned by all operations</td>
</tr>
</tbody>
</table>

Table 25: Comparison of Integrated and Standalone Versions of the CC2420 Radio Transceiver

An incremental approach to porting the radio was taken, thereby limiting the number of possibilities that may have caused problems. First, the radio implementation was decoupled from the SPI bus by replacing \textit{CC2420SpiC}, which is a component in the CC2420’s implementation. It was replaced with a configuration in the platform glue-code directory for the radio, \textless\textit{TOS}\textgreater/cc2430/chips/cc2420, as shown in Figure 44; this directory contains the platform specific implementation components for the radio.

\textit{CC2420SpiC} wires to the connection-specific interfaces used such as \textit{GeneralIO} as it is a physical pin on the standalone version. The initial implementation for the integrated version of the radio will provide these interfaces, even though a lot of the operations are unnecessary. Once this implementation is fully-
functional and has been tested, these interfaces will be substituted for a connection-agnostic interface, shown in Code Sample 43.

Other than removing the use of `CC2420SpiC`, and replacing it with `CC2420ConnectC` in the platform glue-code, the CC2420 chip implementation has been left largely untouched. Ideally it would not have required modification at all.

**Platform Code**

```c
interface CC2420ChipConnection
{
    /* Connection independent commands */
    async command bool getCCA();
    async command bool getSFD();
    async command bool getFIFO();
    async command bool getFIFOP();

    /* RF Interrupt Register */
    /* Connection independent events _time_ is the time the interrupt fired*/
    async event void firedSFD(uint16_t time);
    async event void firedCCA();
    /* Number of bytes in rx fifo greater than threshold */
    async event void rxFifoThresholdHit();
    /* Voltage regulator turned on */
    async event void vregOn();
    /* Packet transmission completed */
    async event void txDone();
    /* CSMA-Ca/strobe processor (CSP) condition true */
    async event void cspWait();
    /* CSMA-Ca/strobe processor (CSP) execution stopped */
    async event void cspStop();
    /* CSMA-Ca/strobe processor (CSP) INT instruction executed */
    async event void cspInt();

    /* RFERR Interrupt Register */
    /* Results from rx fifo overflow or tx fifo underflow */
    async event void fifoError();
}
```

**Code Sample 43: Connection-agnostic Interface - Suitable for Integrated and Standalone versions of the CC2430**

`CC2420ConnectC`, the connection-agnostic version, and `CC2420ConnectC`, the SPI-specific version, provide the same interfaces as it is a direct replacement, as shown in Figure 45. The way in which this configuration is wired however, is completely different. The design of the chip-specific region of the radio abstraction is shown in Figure 46. It should be noted that the existing CC2420 implementation...
expects two other components to exist `HplCC2420PinsC` and `HplCC2420InterruptsC`, which are used by the radio’s receive, transmit, and control components. As mentioned, the initial implementation will modify the platform-agnostic components as little as necessary, however, the final implementation will remove the dependence on these components by employing the connection-agnostic interface presented in Code Sample 43 above.

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Generic Components</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CC2420Fifo</code></td>
<td><code>HplCC2420Register</code></td>
</tr>
<tr>
<td>read(data: uint8_t, out: uint8_t, cc2420StatusT)</td>
<td></td>
</tr>
<tr>
<td>write(data: uint8_t, in: uint8_t, cc2420StatusT)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Generic Components</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CC2420Register</code></td>
<td><code>CC2420Connect()</code></td>
</tr>
<tr>
<td>read(offset: uint8_t, out: uint8_t, cc2420StatusT)</td>
<td></td>
</tr>
<tr>
<td>write(offset: uint8_t, in: uint8_t, cc2420StatusT)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Generic Components</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CC2420Register</code></td>
<td><code>CC2420Connect()</code></td>
</tr>
<tr>
<td>read(data: uint8_t, out: uint8_t, cc2420StatusT)</td>
<td></td>
</tr>
<tr>
<td>write(data: uint8_t, in: uint8_t, cc2420StatusT)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 45: Interfaces and Generic Components Used by the Chip-specific Components of the Radio Abstraction
Figure 46: Design of Chip-specific Region of the Radio Abstraction
Some interfaces that are required for the standalone version of the radio are not applicable to the
integrated version. For example, the strobe commands SXOSCON and SXOSCOFF are not necessary
as the radio does not require an external oscillator. These interfaces are wired to null implementations in
\texttt{HplCC2420SocFakeP}.

The \texttt{CC2420Ram} interface is both connection-agnostic and provides a logical abstraction. However, the
underlying implementation is significantly different to the standalone radio. The radio RAM is actually
accessed through a number of XDATA radio registers, the specific number of which depends on the
length of the data, as shown in Code Sample 44. The access method requires a separate interface to be
defined for each section of radio RAM. The assignments are performed in an unrolled loop for lower
RAM usage and marginally better performance.

\begin{lstlisting}[language=C]
async command cc2420_status_t IEEEADR.write(uint8_t offset, uint8_t* data,
                                           uint8_t len)
{   /* explicitly assigned as more optimal than a loop */
    IEEE_ADDR0 = data[0];
    IEEE_ADDR1 = data[1];
    IEEE_ADDR2 = data[2];
    IEEE_ADDR3 = data[3];
    IEEE_ADDR4 = data[4];
    IEEE_ADDR5 = data[5];
    IEEE_ADDR6 = data[6];
    IEEE_ADDR7 = data[7];
    return GenerateStatusByte();
}

async command cc2420_status_t PANID.read(uint8_t offset,
                             uint8_t* data,
                             uint8_t len)
{   /* explicitly assigned as more optimal than a loop */
    data[0] = PANIDL;
    data[1] = PANIDH;
    return GenerateStatusByte();
}
\end{lstlisting}

\textbf{Code Sample 44: Sample of Radio RAM Implementation}

The radio register interfaces for accessing IOCFG0 and IOCFG1 necessitate being individual for
similar reasons to RAM access. Specifically, the content of each of these registers in the standalone
version is distributed across four registers on the integrated CC2430. This is a significant departure
from the implementation required for the other radio registers.

A generic module was created for accessing the remaining radio registers, as the required actions are
the same. Although passing SFRs as arguments was purposely avoided, because the gain was shown to
not compensate for the associated complexity, XDATA arguments are accessed differently. A similar
\textit{hack} to that used for the Atmega128’s GPIO components is required to cast the “constant” argument to
an assignable address. The implementation gets or sets the 16-bit radio register, which is accessed as
two 8-bit registers in the integrated version. The arguments the component takes are the two 16-bit
XDATA addresses that, between them, contain the 16-bit radio register value.

\begin{lstlisting}[language=C]
generic module HplCC2420SocGenericRegisterP(uint16_t regLowAddr,
                                           uint16_t regHighAddr)
{   provides interface CC2420Register as Reg;
}
implementation
{
    #define regLow (*(volatile xdata *)regLowAddr)
    #define regHigh (*(volatile xdata *)regHighAddr)

    /*************************************************************************
     * Generates a status byte that is compatible with the standalone CC2420's  
     * cc2420_status_t GenerateStatusByte();                             
     */
    async command cc2420_status_t Reg.read(uint16_t* data)
     {   *data = (uint16_t)((uint16_t)regHigh << 8);
         *data |= regLow;
         return GenerateStatusByte();
     }
\end{lstlisting}
async command cc2420_status_t Reg.write(uint16_t data)
{
    regHigh = data >> 8;
    regLow = data & 0xFF;
    return GenerateStatusByte();
}

/***************** Local Function Implementations **************/
/*Generates a status byte that is compatible with the standalone CC2420's*/
c2420_status_t GenerateStatusByte()
{
    cc2420_status_t status = 0;
    status |= READ_BIT(TCON, 1);
    status |= !(READ_BIT(ENCCS, 3));
    status |= READ_BIT(RFSTATUS, 4);
    status |= READ_BIT(FSCTRLH, 2);
    //FIXME - the RSSI value may need to be read to get the RSSI VALID bit
    status |= 0x42;
    return status;
}

Code Sample 45: Radio Register Access - Generic Module

HplCC2420SocP contains the remaining interfaces. Strobe commands and FIFO access is provided
through parameterised interfaces. The FIFO interfaces post tasks to signal the completion of reads and
writes because no hardware interrupt is generated, shown in Code Sample 46, unlike the SPI-connected
version.

async command cc2420_status_t Fifo.write(uint8_t addr)(uint8_t* data, 
    uint8_t len)
{
    address = addr;
    txBufPtr = data;
    txBufLen = len;
    /* enable the tx done interrupt */
    CLR_BIT(RFIF, 6);
    SET_BIT(RFIM, 6);
    txBufPos = 0;
    atomic
    { 
        while (txBufPos < len)
        {
            RFD = data[txBufPos];
            txBufPos++;
        }
    }
    post WriteDone();
    return GenerateStatusByte();
}

task void WriteDone()
{
    signal Fifo.writeDone[address](txBufPtr, txBufLen, SUCCESS);
}

Code Sample 46: Event Signalling through Task Context, because no Interrupt is Generated by Hardware

Two issues were discovered with code compilation by the IAR compiler, one affects all 8051
compilers, and the other may be specific to IAR. The radio’s CSMA (Carrier Sense Multiple Access)
relies on the ability to perform random duration back-offs; the implementation of which utilises 64-bit
integers and performs a 32 right-shift. The issue with this is the syntax translation script will modify
the 64-bit variable to a 32-bit type, always resulting in a zero duration once the bit-shift has occurred.
The second issue is caused by the IAR compiler’s inability to interpret the shorthand syntax used in
CC2420TransmitP, shown in Table 26. Both cases require modifications to core radio subsystem
components to overcome.

Original Implementation:
uint8_t tx_power = getMetadata( m_msg )->tx_power;

Modification Required:
cc2420_metadata_t* msg_metadata = getMetadata( m_msg );
uint8_t tx_power = msg_metadata->tx_power;

Table 26: Shorthand Syntax the IAR Compiler Cannot Interpret & the Modifications Required
MIG (Message Interface Generator for TinyOS), which is a tool for generating code to process TinyOS messages from the message type specified in a TinyOS application, was not capable of parsing the header file containing the SFR definitions. The workings of MIG were not investigated, as supplying it with pre-processed data allowed it to compile the Java application for the host, before continuing to compile the TinyOS application for the target.

The integrated version of the radio has slightly different behaviour than the discrete version. Furthermore, these differences are not documented. For example, the behaviour of the CCA (Clear Channel Assessment) status flag is different at radio startup, which is the trigger for signalling to the application that the radio is ready to use. The integrated version of the radio provides a CCA interrupt, the functionality of which is different to the CCA status flag. However, even emulating the interrupt with time-delayed polling in the same way as the standalone implementation does not generate the same actions. Due to the massive intrusiveness of the debugger, discussed in Appendix C, and the only reference of the standalone version’s behaviour being the TinyOS source, it is not possible to determine how the integrated radio’s behaviour differs from the standalone version.

Debugging the radio is extremely challenging due to the tight timing constraints of certain operations. This issue was further complicated because only one end of the RF link could be debugged, because only one evaluation board was available. Additionally, the development board is required to flash the CC2430. An 802.15.4 dongle was employed [101], which provides packet sniffing capabilities. This device was essential in successfully implementing radio transmission support.

12. Future Work

12.1 Fully Automated Integration of the IAR Compiler

Currently compilation of TinyOS applications is a two stage process. Whilst this has proven sufficient for development, fully automating the build process would make the platform slightly easier to use.

12.2 Further Energy Efficiency Optimisations

Slight energy efficiency gains should be possible through modifications to some subsystems.

12.2.1 MCU Clock Speed

Currently the MCU is allows clocked at 32MHz, however it may be possible to improve energy efficiency by clocking the MCU at 16MHz while the radio is not required, despite the general lifecycle of a WSN application (Figure 1) which performs very little work before using the radio. Lynch and O’Reilly make a similar claim, and acknowledge that it is dependent on the application in question [30].

An interface which allows the clock speed to be safely altered at runtime would allow the effects of this, on applications with various demands, to be realised. Consideration must be given to the effective timer frequencies and peripheral bus (e.g. UART) baud rates.

12.2.2 Direct Memory Addressing (DMA)

Use of the DMA controller for high-frequency or large data transfers would increase energy efficiency. By implementing support for DMA transfers the level of energy saved for various applications could be realised.

A significant number of MCUs are equipped with DMA controllers, which have roughly equivocal functionality, to warrant a DMA subsystem. Due to the chip-specific nature of these controllers (e.g. number of channels, what peripherals the channels can be assigned to) a weak HIL a la the ADC subsystem would be necessary. Further, because application requirements may not suit using DMA, such as when small quantities of data are being transferred, it should be possible to use the orthodox method. This can be achieved using SIDs (Source and Sink Independent Drivers) interfaces [122], which can be utilised to present a common interface to the top-level of a subsystem that employs DMA.
An additional level of indirection would be introduced because a translation between interfaces would be required. On completion, testing would be essential to ensure the cost of the additional level of indirection did not exceed the potential gains.

12.3 GPIO & GPIO Interrupts

Although it is advantageous to separate the GPIO and GPIO interrupt functionality into two separate subsystems both are often required for a particular pin, therefore a configuration that wired to both of these subsystems would be convenient; this would simplify the wiring for the developer.

12.4 GPIO Interrupts

The architecture of the CC2430’s GPIO interrupts limits the selection of which edge an interrupt will be generated on to a per-port basis, rather than a per-pin basis. The implications of this are discussed in Section 11.9.1, but the CC2430 device’s GPIO is still useable in the vast majority of situations.

The current implementation passes responsibility to the system architect to ensure the edge-trigger of pins for a particular port do not conflict, however there is the potential for error on the developer’s part which would result in a particular awkward bug to diagnose. By introducing a resource arbiter at the port level that allows pins to be used in a port provided they request the same interrupt edge-trigger would provide runtime prevention and easy detection of this misuse. Through the use of nesC annotations and an edge-trigger misuse script, whose invocation would need to be added to the CC2430s makefile, the application could be analysed at the time of compilation and generate warnings about potential edge-trigger conflicts on a specific port.

12.5 Radio Receive Support

With radio receive support this port to the CC2430 will be a fully useable TinyOS-2.x platform for WSN application development. This will also allow research into heterogeneous mote inter-communication.

12.6 Heterogeneous Mote Communications

To date there has been very little published work on inter-communication of heterogeneous devices in wireless sensor networks, brought about in part by limitations in TinyOS-1.x [78].

All TinyOS platforms with 2.4GHz radios utilise the discrete CC2420 chip. Despite the CC2430’s radio essentially being the same it is a variant, and will allow experimentation with heterogeneous radio hardware.

12.7 Energy Efficiency Comparison with Other Mote Platforms

It is desired to perform an energy efficiency comparison of the CC2430 with other mote platforms. The results yielded from this research will further clarify the suitability of the platform, and provide an insight to the level of gains brought from second generation integrated hardware.

13. Evaluation & Conclusion

13.1 Appraisal of the CC2430

From performing this work it is apparent that a device that appears to be an ideal candidate as a TinyOS platform may not have the hardware facilities (what interrupts are generated and when) expected by and, suited to, the architecture of TinyOS; the devil is most definitely in the detail.

Specifically, core interfaces expect the hardware to generate interrupts for certain events and expect certain features to exist, neither of which are sometimes satisfied by a particular platform. This work
has shown that these issues can often be worked around by emulating features in the software implementation, though this is to the detriment of the devices energy efficiency.

The implications of the hardware implementation limit the suitability of the CC2430 for WSN applications. Furthermore, where the device’s functionality has not met the assumptions made by the TinyOS architecture, software emulation has been necessary; further diminishing the chip’s performance. The consequences of these, and whether the gains compensate for these, require an energy efficiency comparison to be undertaken; before this has been performed a true evaluation of the CC2430 cannot be carried out.

The results of an energy efficiency comparison are expected to conclude that the CC2430 offers similar performance to the Telos platform, the mote with the greatest performance currently available. However, the CC2430 provides the benefits of a SoC device; it has a much smaller footprint, a lower BOM, and requires a less complex circuit design. Until experimentation is undertaken however, this is speculative.

It is the author’s claim that in the present climate the Chipcon CC2430 device is a suitable platform for WSN applications. With a fully functional radio and the proposed modifications to TinyOS and nesC the CC2430 would be a capable TinyOS platform for a large range of WSN applications, provided the application in question does not exceed the limitations discussed in this document.

Additionally, the reliance on a commercial compiler is not desirable. It is hoped that SDCC will be a viable option in the future.

13.2 Contributions

A number of contributions have been made as a result of this work:


This paper outlines the issues involved and difficulties encountered porting TinyOS to the CC2430 System-on-Chip platform. Core issues that limit the portability of TinyOS-2.x are identified, and suggestions made as to how these might be addressed. Specifically, a new directory structure is proposed which allows component reuse and organizes devices into chip-families, considerations for System-on-Chip devices, and extensions to nesC.

The author is a member of the TinyOS 8051 Working Group.

The source code for this project has been made publicly available in the TinyOS source repository, released under the TinyOS license, at [http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/nixtems/](http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/nixtems/).

13.3 Were the Project Aims Met?

On starting this project I had an idealised set of aims which I wished to accomplish. However, this is *real* research and it was fully acknowledged that porting TinyOS-2.x may not have been possible; neither an 8051-cored processor nor an integrated mote platform had been ported to TinyOS-2.x.

With the exception of radio receive support, this work represents a fully useable port of the CC2430 platform. Additionally, the findings from this work offer a clear unbiased assessment of the device’s suitability as a WSN platform.

When the above is considered an incredible amount has been accomplished. The lack of a fully functional radio prevents the full power of TinyOS from being harnessed. However, by employing a device for use as a base station, such as the Integration 802.15.4 radio dongle [101], it is possible to develop star topology WSN applications.
13.4 Personal Reflection on the Project

I feel that much has been gained from undertaking this project. This project has given me an opportunity to research the field of Wireless Sensor Networks, which is still in its infancy. Starting from a point where I knew very little, I have managed to understand the research area and port TinyOS-2.0 to a second generation integrated device.

On embarking on this project the complexity of the subject matter was not fully appreciated. The depth and breadth of research undertaken has been far greater than anticipated. Whilst this has sometimes been more deeply than required, this has resulted in a greater understanding of the influences on WSN research, and provided a clear picture of the future evolution of WSN platforms, software, and systems.

I have learnt an incredible amount about the internal workings of TinyOS and the actions performed by the nesC compiler. I believe that optimal processor selection requires an intricate knowledge of the underlying TinyOS architecture, as small details have potentially large effects in terms of energy efficiency.

Having undertaken a port to a new architecture I am aware of the implications caused by TinyOS’ assumptions. However, the aim of TinyOS-2.x reducing complexity from an applications development perspective has masked the importance and level of understanding required. Additionally, slightly different implementation approaches can have significant effects on the efficiency of the abstraction.

A lot of the knowledge acquired is extremely pertinent to all embedded systems development. Energy efficiency and designing a system which can be implemented in an optimum way for devices with constrained resources. Whilst the vast majority of embedded systems do not have such severely constrained resources, working with WSN devices ensures that the developer understands how code size, RAM usage and power consumption are affected by various implementation paths. This allows device specification for a project to be performed confidently.

The contributions from this project have benefited the WSN community, which gives me enormous satisfaction.

I am extremely satisfied with the accomplishments that have been achieved. It would have been nice to have got the radio transceiver fully functional, as this would make the CC2430 a more useable development platform for WSN applications. Nonetheless, I am pleased with the completeness of the port, considering the complications and initially unseen implications caused by:

- Architecture and functionality assumptions made by TinyOS-2.x, which the CC2430 fails to meet.
- Issues found with the hardware, specifically where it does not function correctly.
- Lack of sufficient information and omission of details in datasheets and TinyOS documentation.

The iterative development lifecycle employed for this project has proved to be both effective and suitable. If I were to undertake this work with the knowledge I now possess, the design decisions and implementation directions would lead to better solutions from the off set. For example, the millisecond timer abstraction originally used Timer1 as its underlying hardware counter, it was not until further information was discovered that the implications of this were realised; resulting in its reimplementation.

Another example lies with the detailed understanding of the nesC compiler’s actions. Which were discovered post-implementation through inspection of the generated C source. With the knowledge I now possess the final implementation of the GPIO subsystem performed in this project, would be the first implementation realised.

Additionally, I would ensure that sufficient hardware and test equipment was available when required before undertaking a project. Because, as the radio has proved, the lack of these can result in substantial time being consumed trying to discover and rectify problems.

My over-ambition has driven me to demand far more from myself then anticipated, in terms of the work performed, the scope of research, and understanding, which has taken substantially longer than my original estimates. However, the result has been greater than I should have realistically hoped to achieve. I have found the sheer illusion of accomplishing a goal, especially when its timely pursuit
seems hopeless, is a massive motivator; that feeling generates the drive, enthusiasm and commitment required to overcome even the greatest of obstacles.

Since completion I have improved my time-estimation abilities, due largely to the lessons learnt from undertaking this project. The implications of small details must not be under-estimated, as they can often amount to substantial effort being required, and potentially prevent a device or software framework from being suitable for the targeted purpose.

### 14. Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>802.15.4</td>
<td>IEEE standard for low-power low-data-rate radio communications in the 2.4GHz range.</td>
</tr>
<tr>
<td>Chip</td>
<td>In TinyOS a chip is generally a single silicon component, however SoC devices such as the Chipcon CC2430 are blurring this definition, as it consists of an 8051 core and an a Chipcon CC2420 radio.</td>
</tr>
<tr>
<td>HAA</td>
<td>Hardware Abstraction Architecture.</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware Adaptation Layer.</td>
</tr>
<tr>
<td>HIL</td>
<td>Hardware Interface Layer.</td>
</tr>
<tr>
<td>HPL</td>
<td>Hardware Presentation Layer.</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>MCU</td>
<td>Micro Control Unit.</td>
</tr>
<tr>
<td>Mote</td>
<td>The name that wireless sensor network devices have been coined with.</td>
</tr>
<tr>
<td>nesC</td>
<td>Network Embedded Systems C language; an object orientated extension to the C programming language.</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System.</td>
</tr>
<tr>
<td>Platform</td>
<td>In TinyOS a platform is a mote type.</td>
</tr>
<tr>
<td>RF</td>
<td>Radio Frequency.</td>
</tr>
<tr>
<td>Sensorboard</td>
<td>A board, providing additional sensors and other peripherals, which the mote can be connected to.</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network.</td>
</tr>
</tbody>
</table>
Appendix A - Discussion of 8051 Compiler Options

There are a large number of commercial 8051 compilers on the market, which have similar features, but the comparison presented here is limited to the two predominant compiler suites, Keil and IAR, plus SDCC an open source C compiler for a number of small MCU architectures including the 8051.

1. Keil Compiler

The Keil compiler has been usefully employed by the TinyOS 8051 Working Group [102] and WISENET [103], in their efforts to port TinyOS-1.x to an 8051-cored platform. The syntax translation script used by these groups is available and the compiler proven. However both platforms’ radios were sub-2GHz radios, which were not 802.15.4 compliant and had far less stringent timing requirements and [102] did not implement the radio components. The relevance of this is that TinyOS relies on function inlining to remove the impact of the large number of function calls between components, which can amount to a significant overhead; unlike GCC the Keil compiler does not have this capability.

The Keil compiler includes explicit support for the Chipcon CC2430; this is of note as the CC2430 contains an enhanced 8051 processor core. A code-size limited trial is available for download from Keil [104]. It is available for Microsoft Windows only.

2. IAR Compiler

The IAR compiler is recommended by Chipcon and has explicit support for the CC2430; the demo application, which is shipped with the development kit, is provided as source code for the IAR compiler. A time-limited trial with full support is supplied with the CC2430 development kit, and a code-size limited trial is available for download from IAR Systems [105]. It is available for Microsoft Windows only.

In extended mode this compiler has the ability to perform C function inlining, which is not ANSI or ISO C compliant, but is performed by GCC. TinyOS is reliant on this capability to remove the impact of the large number of function calls made between modules, thereby ensuring that time-critical operations, such as radio packet transmission, are performed correctly [16]. Function inlining also reduces power consumption, which is of key importance for WSNs, by reducing the number of machine cycles, thereby allowing the MCU to be put into sleep mode, which consumes less power, for longer periods (see Figure 4).

Having performed some preliminary experimentation with the CC2430 and IAR compiler suite, the interactive debugger was proven to be functional and appeared comprehensive. How debugging will be performed is an important issue to address from the very beginning of a project, none more so than OS ports to embedded devices.

There is no evidence of use of the IAR 8051 compiler suite with TinyOS, either past or present.

3. SDCC – Small Device C Compiler

SDCC [106] is the only 8051-capable open source compiler. Ideally this compiler would be used as a substitute for GCC, as it is released under GPL and can therefore be freely distributed without charge, whereas licenses for both the Keil and IAR compilers are thousands of pounds. SDCC is available for Microsoft Windows, Linux, and Mac OS X, increasing choice of development platform.

This compiler has been used for the RISE project [107] performed in 2005, however this contradicts the findings of the TinyOS 8051 Working Group (T8051WG) who claim that SDCC is not suitable for TinyOS development [102]:
In our experience the SDCC compiler and associated tools are not yet mature enough to support our development. We recommend pursuing other alternatives such as KEIL or other compiler suites.

They discovered a number of issues with the compiler, and found the debugger (SDCDB) was not functional. Considering that T8051WG undertook their work in March 2006, after RISE, and they found some fundamental type signed-ness bugs (which have since been fixed) coupled with the lack of RISE source in the public domain leads the author to question the authenticity of RISE’s claim.

Explicit support for the CC2430 was added to SDCC 2\textsuperscript{nd} December 2006 [108], long after development had started. Unlike GCC, SDCC has no function inlining capabilities.

It should be noted that SDCC is being actively developed, and it is hoped that this compiler and debugger will be a viable alternative in the future.

4. Decision & Justification

The author chose to use the IAR compiler suite for the port of TinyOS to CC2430. All three compilers have a similar syntactical dialect [109] [110] [111], especially the SDCC and Keil compilers, however there are differences that prevent migration to an alternative compiler without alterations to the source.

The Keil compiler would be the option with least risk; it has already be proven with TinyOS-1.x ports and a syntax translation script is available, however the IAR compiler should be capable of generating more efficient code thanks to its function inlining.

Despite the slightly higher risk, for the above reason, and those presented in Section 11.1.2, the author pursued the use of the IAR compiler, which had not been used in previous TinyOS porting attempts, in order to fully realise the effects of an unsupported compiler suite. The differences that exist between compilers for the same processor architecture will tie a chip and platform implementation to a single compiler, which is not desirable and needs to be addressed by TinyOS and nesC.

SDCC was eliminated as a possible compiler choice, despite the claim by RISE [107], because the TinyOS 8051 Working Group stated that it was not mature enough [102], and there is no tangible evidence that it is, at the time of writing, suitable for TinyOS development on an 8051-cored platform. Combined with the fact that SDCDB is not functional (therefore no debug facility), the CC2430 was not explicitly supported at the start of development, and Ian Johnson’s troublesome experiences trying to use SDCC on the raw CC2430 hardware, the author concludes that it is not, currently, suitable.
Appendix B - Creating a New TinyOS Platform & The TinyOS Build System

The TinyOS build system utilises the GNU make utility supported by Perl script insertions. This allows machines with Linux, Mac OS X, and Microsoft Windows (in conjunction with Cygwin) operating systems to be used for TinyOS development.

A number of file extensions are used to denote the function that the file has within the build system, described in Table 27 below.

<table>
<thead>
<tr>
<th>File Extension</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.target</td>
<td>Defined for each platform. Enables platform support within TinyOS, contains platform specific options and includes the processor-core’s rules file. The <code>&lt;platform&gt;.target</code> file is selected by the target specified to <code>make</code>: $ make <code>&lt;platform&gt;</code></td>
</tr>
<tr>
<td>.rules</td>
<td>Defined for each processor-family. Contains rules &amp; chip-family specific options for the processor family. This file contains the actual <code>make</code> targets for the processor.</td>
</tr>
<tr>
<td>.extra</td>
<td>These are optional compilation rules that can be included into the build system on the command line. Multiple options can be specified. The <code>&lt;option&gt;.extra</code> file is selected by specifying the option to <code>make</code>: $ make <code>&lt;platform&gt;</code> <code>&lt;option&gt;</code></td>
</tr>
<tr>
<td>.platform</td>
<td>Contained in the platform’s implementation directory, e.g. <code>&lt;TOS&gt;/platforms/&lt;platform&gt;/platform</code>. This is mandatory for each platform. It contains Perl snippets, which include directories and set compiler options, that are interpreted by ncc.</td>
</tr>
<tr>
<td>.sensor</td>
<td>Contained in the sensorboard’s implementation directory, e.g. <code>&lt;TOS&gt;/sensorboards/&lt;sensorboard&gt;/sensor</code>. This is mandatory for each sensorboard, though can be empty. It contains Perl snippets that are interpreted by ncc, which can add or modify any compile-time options necessary for a particular sensor board.</td>
</tr>
</tbody>
</table>

Table 27: Types of Files in the TinyOS Build System: Their File Extensions & Roles

The build system is located in `<TINYOS>/support/make`, which contains supported compilation options for target platforms, generic options and make rules. There is a subdirectory for each processor family, which contains chip-family specific and programmer options.

A description of the steps to perform for a standard TinyOS installation are provided by [112].

The ability to use Microsoft Windows as a development system enables the use of compiler suites only available for Windows, such as Keil and IAR. To invoke these from within Cygwin, so that it can be used as the sole development environment requires the path to the compiler to be exported to Cygwin’s `PATH` environment variable, as in Code Sample 47 below; this has been added to the development user’s `.bashrc` file to automatically export the environment variable on instantiation of the bash shell.

```
bash$ export PATH=$PATH:"/cygdrive/c/Program Files\IAR Systems\Embedded Workbench 4.05 Evaluation version\8051\bin"
```

Code Sample 47: Exporting Path to Windows Program for Invocation from Cygwin

To date the platforms supported by the mainline TinyOS-2.x distribution are all unintegrated motes (i.e. MCU plus radio). As discussed earlier in this document, the CC2430 is a SoC device that consists of an enhanced 8051 processor core and a CC2420 radio transceiver. The purpose of differentiating between chip and platform code by TinyOS-2.x is to reduce code duplication, which existed in previous versions of TinyOS, therefore the CC2430 has been treated as a TinyOS platform that consists of an 8051 chip and a CC2420 radio chip.
It should be noted that the separation of code that implements chip, platform and sensorboard components is one of the many changes between TinyOS-1.x and TinyOS-2.x; therefore partial ports of TinyOS-1.x to SoC platforms, such as the CC1010 by WISENET [103] and the nRF24E1 by T8051WG [102], did not have these considerations.

1. Adding Support for a New Platform

TinyOS platform implementations are responsible for tying the implementation of the chips that they consist of together, plus any features or peripherals which are platform dependent. For example the chip connection code

Support for new mote platforms is added to TinyOS through the addition of a `<mote_platform>.target` file to `<TOS>/support/make`, namely `cc2430.target` in the case of the CC2430 (Code Sample 48). The `$(call TOSMake_include_platform,mcs51)` line includes the platform file for the 8051/mcs51 family of processors, namely `mcs51.target`. Important extracts of which are provided in Code Sample 49.

```
PLATFORM=cc2430
MCS51FLAGS=
NESEC_FLAGS=-Wnesc-all
$(call TOSMake_include_platform,mcs51)
cc2430: $(BUILD_DEPS)
@:
```

*Code Sample 48: TinyOS Target File to Include the CC2430 Platform into the Build System - cc2430.target*

The first extract varies from standard build rules, used on platforms with GCC support, in that it informs the build process to stop after the generation of the C file; effectively after the nesC compilation and before the target C compiler.

The compilation rule `exe0`, shown in the second extract of Code Sample 49, consists of two invocations of ncc. The first compiles the TinyOS application, from nesC, to a single C file (the `-conly` option in the first extract achieves this, but requires the C file to be copied to the build directory). The second invokes the syntax translation script and target compiler, to generate the executable.

```
PFLAGS += -Wall -Wshadow -DDEF_TOS_AM_GROUP=$(DEFAULT_LOCAL_GROUP) $(NESC_FLAGS)
PFLAGS += -target=$(PLATFORM) -conly -board=$(SENSORBOARD)
ifdef MSG_SIZE
PFLAGS += -DTOSH_DATA_LENGTH=$(MSG_SIZE)
endif
...
exe0: builddir $(BUILD_EXTRA_DEPS) FORCE
 @echo "    compiling $(COMPONENT) to a $(PLATFORM) binary"
 @echo "    compiling $(COMPONENT) to a $(PLATFORM) binary"
 @cp $(COMPONENT).c $(BUILDDIR)/app.c
 $(NCC) -v -o $(MAIN_EXE) $(MCS51FLAGS) $(OPTFLAGS) $(CFLAGS) $(WIRING_CHECK_FLAGS) $(BUILDDIR)/app.c $(LIBS) $(LDFLAGS)
ifdef WIRING_CHECK_FILE
 @nescc-wiring $(WIRING_CHECK_FILE)
@end
```

*Code Sample 49: Extracts from the 8051 Processor Compilation Rules - mcs51.rules*

The compilation rules file for the 8051 processor core employs an additional `MCS51FLAGS` variable: `MCS51FLAGS`, which contains 8051 specific options. This variable is set by `iar.extra` (Code Sample 50), which was created to allow the target compiler, and its associated options, to be specified on the command line. Support for alternative compilers can be added through the creation of other `<compiler>.extra` files such as `keil.extra` or `sdcc.extra`.

```
#-*-Makefile-*- vim:syntax=make
MCS51FLAGS = -D__IAR_SYSTEMS_ICC__ -specs $(TINYOS_MAKE_PATH)/mcs51/iarspecs.txt
```

*Code Sample 50: IAR Specific Options - iar.extra*

The `.platform`, which must exist for every platform contains Perl snippets that include the platform’s relevant directories. It is pulled in by the build system, which obtains the target platform from an argument to `make`. 
A number of components are expected by TinyOS, which support the core functionality and common subsystems. The platform configuration, \textit{PlatformC.nc}, and module, \textit{PlatformP.nc}, source files, and \textit{hardware.h} are mandatory for all TinyOS mote platform. Other HIL-layer components, such as \textit{HilTimerMilliC} and \textit{PlatformLedsC}, are required in order to utilise the platform-independent functionality provided by their respective abstractions.

\section*{2. Adding Support for a New Chip}

The following is stated in \cite{113}, which is a sentiment the author shares:

\begin{quote}
\textit{Developing drivers for a new chip can be a non-trivial undertaking (especially for radios and microcontrollers).}
\end{quote}

It should be noted that a chip port is always accompanied by a platform port. This is because the boot sequence and scheduler wire to mandatory components at both the chip and platform level; this is discussed in \cite{79} and Section 11.7. A component named \textit{McuSleepC} is expected in the \textit{chips} directory, which is responsible for establishing the lowest possible power mode that can be used at the time its operations are called. This component is at the heart of the default scheduler.

\subsection*{Code Sample 52: Additional Rules for use of TinyOS with the IAR Compiler - iarextra}

\begin{verbatim}
MCS51FLAGS = -D__IAR_SYSTEMS_ICC__ -specs $(TINYOS_MAKE_PATH)/mcs51/iarspecs.txt
\end{verbatim}

\section*{3. Adding Support for a New Sensorboard}

Adding support for a new sensorboard in TinyOS requires a directory for the sensorboard to be added to the TinyOS source tree; the standard location for this is \texttt{<TOS>/sensorboards/<sensorboard>}. The TinyOS build system expects to find a file named \texttt{.sensor} in the sensorboard directories \cite{114}, which may contain sensorboard specific compilation options, or be empty; as is the case with the RF04EB sensorboard, a component of the CC2430 development kit.
The sensorboard can be specified at compile time by using a statement like that in Code Sample 53. An alternative solution is to define an additional `.extra` file, which defines SENSORBOARD; this is the approach taken by the author, which can be seen in Code Sample 54.

```
SENSORBOARD=<sensorboard name> make <mote>
```

**Code Sample 53: Selection & Inclusion of Sensorboard**

```
SENSORBOARD=RF04EB
```

**Code Sample 54: Use of `.extra` File to Select Required Sensorboard – rf04eb.extra**

### 4. Adding a New Application

From the perspective of supporting the build system an application is responsible for supplying only a single auxiliary files, namely it must provide a makefile. An example makefile for the Blink application is presented in Code Sample 55; it is very simple as the majority of the work is performed by the `Makerules` file that it includes, however additional compilation options can be added to a makefile if necessary.

```
COMPONENT=BlinkAppC
include $(MAKERULES)
```

**Code Sample 55: Example TinyOS Application Makefile**
Appendix C - Debugging TinyOS

Debugging an embedded device that one is porting an operating system (OS) to, immaterial of which OS, is always challenging. In the early phases of the port there is no support for any peripherals; not even an LED can be blinked. The developer must rely on methods which allow direct access to the MCU, such as JTAG, In-Circuit Emulators (ICEs) or In-Circuit Debuggers (ICDs).

Debugging TinyOS applications is incredibly awkward, but particularly so when porting TinyOS to an unsupported chip architecture. The C code that the nesC compiler generates is obfuscated due to:

- The static instantiation of objects.
- The optimisation performed by the nesC compiler.
- The name-mangling of variables and functions performed in order to express the object-oriented like program in C, a procedural language.
- Code from all components (and header files) in the application’s call graph results in a single C file.

An insight into the extent of this is presented by the comparison of nesC to the generated C code in Code Sample 56, where nesC is black and C is red. The affect the nesC compiler has on generic interfaces, component scope member variables, and parameterised interfaces is shown in Code Sample 57.

For an architecture with GCC support the generation of a single source file is irrelevant, however this makes debugging more awkward when using an unsupported toolchain. With the supported GNU tool suite GDB can be used to debug at nesC source level; the only slight complication being that the mangled function and variable names must be used when specifying breakpoints, as described in [115]. However, there are no GNU tools for 8051-cored MCUs; the IAR tool suite must be used, which is only capable of C source level debugging. The generated C file for even a relatively simple application is very large (Blink shown in Code Sample 56 generates a ~3000 line C file), adding to the difficulty of debugging.

```c
#include "Timer.h"

module BlinkC
{
    uses interface Timer<TMilli> as Timer0;
    uses interface Timer<TMilli> as Timer1;
    uses interface Timer<TMilli> as Timer2;
    uses interface Leds;
    uses interface Boot;
}

implementation
{
    event void Boot.booted()
    {
        call Timer0.startPeriodic( 250 );
        call Timer1.startPeriodic( 500 );
        call Timer2.startPeriodic( 1000 );
    }
    //#line 53
    //# 52 "BlinkC.nc"
    static inline void BlinkC__Boot__booted(void)
    {
        BlinkC__Timer0__startPeriodic(250);
        BlinkC__Timer1__startPeriodic(500);
        BlinkC__Timer2__startPeriodic(1000);
    }
    //# 49 */opt/tinyos-2.x/tos/interfaces/Boot.nc"
    inline static void RealMainP__Boot__booted(void){
        //#line 49
        BlinkC__Boot__booted();
        //#line 49
    }

    event void Timer0.fired()
```
Variable name-mangling performed for component scope member variables shown in red. This is an instance of the TransformAlarmC generic component for the first virtualised millisecond timer, shown in blue.

Virtualised Millisecond Timer with 3 Instances – From Blink application shown in above code sample.

Code Sample 56: A Section of a TinyOS Application – nesC in black, generated C code in red
Code Sample 57: Example of Variable Name-mangling, Generic Component-mangling and Parameterised Interfaces

Debugging TinyOS applications can be performed in a number of ways, as described in Table 28:

<table>
<thead>
<tr>
<th>Debug Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOSSIM Simulator [116]</td>
<td>Simulation of TinyOS application run on a PC; executes the same code. The most frequently employed method of debugging TinyOS applications.</td>
</tr>
<tr>
<td>OASIS [117]</td>
<td>Printf() style debugging, through use of UART.</td>
</tr>
<tr>
<td>VDB [118]</td>
<td>Print out debug messages to UART, assertions, and stack checking.</td>
</tr>
<tr>
<td>VWdeBug [119]</td>
<td>Configurable Printf() style debugging over UART. Allows runtime configuration of logging levels.</td>
</tr>
<tr>
<td>Marionette [120]</td>
<td>Query-based poke &amp; peek, Embedded RPC debugging. Utilises extraction of nesC attributes at compile time for use by an external application.</td>
</tr>
<tr>
<td>Native TinyOS printf()</td>
<td>Recently added to TinyOS-2.x, see Section 11.13.1 for more details.</td>
</tr>
<tr>
<td>JTAG/ICE/ICD [115]</td>
<td>Provides low-level, direct chip access, supporting interactive debugging. JTAG allows nesC source level debugging with GDB where platform support exists. The only option in the early stages of an operating system port.</td>
</tr>
</tbody>
</table>

Table 28: TinyOS Application Debugging Techniques

None of the above methods, with the exception of JTAG/ICE/ICD, are suitable for architecture and platform level ports. Simulation caters well for higher level application and routing issues, but does not help with hardware specific problems. Other debugging tools and techniques described above that provide the capability to debug an application on the hardware are reliant on UART and radio communications being available, which is not possible at the start of development with an unsupported chip.

The RF04EB development board enables the CC2430 to be interactively debugged from the IAR IDE; it is effectively an ICD. The CC2430 has three hardware breakpoints, which was found to be marginally sufficient. As TinyOS is event driven, execution flow starts from interrupt requests or tasks; therefore when debugging complex interactions a number of breakpoints are necessary to detect the order of events.

Viewing and modifying SFRs is performed in a poke-and-peek fashion from the debug executable. Use of the debug executable significantly affects execution speed (an example of this is discussed for the UART in Section 11.13). However, the crucial issue is that timer incrementation is not disabled by the debug handlers; this makes any timing-related issues virtually impossible to debug, as the timer’s counts will increment. For example, a change in the sleep timer’s count representing tens of seconds has been witnessed from poking its comparator.
Appendix D - Chipcon RF04EB Sensorboard

The RF04EB is the sensorboard supplied with the Chipcon CC2430 development kit [121]. It is shown in Figure 47 and, when used with the CC2430 evaluation module, has the following facilities:

- LCD Display.
- 2x16 Character.
- Digital Joystick, but uses a potential divider to reduce pin count.
  - Requires treatment as an analogue device and therefore the ADC.
- RS-232 Connector.
- 2 LEDs.
  - 2 of the 4 LEDs on the sensorboard are not wired to the CC2430.
- Potentiometer
- Push Button
- CC2430 I/O Pins are exposed by two headers.

![Figure 47: RF04EB Evaluation/Sensor Board [121]](image)

The schematics of the RF04EB are available in Appendix E.

1. Design & Implementation

1.1 Push Button

The push button on the RF04EB is not debounced, requiring this operation to be performed in software. The button is connected to one of the CC2430’s GPIO pins. The abstraction’s design utilises SIDs.
(Source and Sink Independent Drivers) interfaces [122]. These interfaces offer little advantage over the GeneralIO interface in this particular scenario, however an exploration of their use was desired.

The SIDs TEP [122] states the requirement for sensor-independent interfaces:

*Applications often build on top of more general systems, such as management or database layers, which may need to sample sensors. Since these are general and sensor-independent systems, they require a sensor-independent interface.*

While this is true, there are a number of other situations that can benefit from SIDs interfaces. For instance platform-independent library components that handle protocols can be written which are independent of the underlying bus: such as NMEA 0183 from a GPS receiver whose underlying connection may be SPI, a UART, or Compact Flash for example. Subsystems that optionally contain other subsystems are another example where SIDs interfaces are beneficial, a discussion of this is presented in 12.2.2 with DMA being the sub-subsystem context.

The design of the user button abstraction is shown in Figure 48. UserButtonC is responsible for performing all wiring of the chip-, platform-, and sensorboard-agnostic interfaces. While HplUserButtonC is responsible for wiring to the actual GPIO pin, the button is connected to the sensorboard. The only implementation module is UserButtonLogicP, which performs the necessary debounce logic (Figure 49). The use of HIL and HPL layer components allows the abstraction to be implemented in a platform-independent manner. This is shown by the way this is wired: UserButtonC wires UserButtonLogicP to HplUserButtonC without needing to know the underlying GPIO pin that the button is connected to.

![Diagram of user button abstraction](image_url)

*Figure 48: Push Button Abstraction*
1.2 LCD

The LCD panel is a SMBus device which is connected to two GPIO pins on the CC2430 which act as clock and data lines. As the CC2430 does not have an SMBus or I²C controller the LCD must utilise the software emulated implementation of the bus controller described in Section 11.11.

The design of the LCD abstraction is presented in Figure 50, the `Lcd` interface provides a number of operations allowing its full functionality to be accessed, whose implementation is realised by the LcdP module which performs all of the necessary SMBus actions. LcdC is the top-level configuration for the LCD device, which instantiates a bit-banged SMBus controller and wires its clock and data lines to the GPIO pins that the LCD panel is connected to, as it is the only component that knows which pins on the CC2430 the LCD is connected to.

![Figure 50: Design of the LCD Abstraction](image-url)
This implementation is sufficient as the LCD is the only SMBus component on the RF04EB sensorboard, however the SMBus protocol supports multiple devices on a single bus which is not catered for with this particular component configuration. If a sensorboard has more than one SMBus component it should have an SmbusControllerC component for each bus that the sensorboard provides, which instantiates the SMBus controller component (SoftwareSmbusC) and provides resource arbitration to the bit-banged SMBus, as shown in Figure 51.

This setup requires an Smbus header file that defines a string to use with unique(), that should be used by each client of the SmbusControllerC component (e.g. DummySmbusDevice1 & DummySmbusDevice2 in this diagram)

![Diagram of SMBus configuration](image)

Figure 51: Configuration Required for a System with Multiple SMBus Devices

1.3 Potentiometer

The potentiometer provided a controlled means of testing the ADC, as the dial can be positioned at any place and the voltage will remain the same. A multimeter was used to obtain the voltage across the potentiometer at a particular position by probing the board, which could be compared to the transformed reading from the ADC to ensure they match. This proved that the ADC subsystem performed correctly.

Through a comparison of the potentiometer and joystick designs (Figure 52 and Figure 55 respectively) it can be seen that the ADC subsystem is very effective at masking the complexity from clients of the subsystem. It should be noted that, although shown below the ADC subsystem component in Figure 52, PotP is actually a higher-level component; it is expressed in this way to show that PotP provides an interface that the ADC abstraction uses. Specifically, the configuration, PotC, wires an interface used by ADCReadClientC to the PotP module which provides it, effectively acting as a callback as shown in Figure 53.
Figure 52: Design of the Potentiometer Abstraction
The top-level configuration for the device which uses the ADC provides the interfaces that the application wires to and wires the internal components of the device’s abstraction.

The device’s private module contains the configuration details for the specific device. It is at the same level as device’s configuration (e.g. sensorboard).

AdcDeviceC

Adc DEVICE P

Adc Subsystem

The top-level configuration wires the ADC subsystem to device’s private module containing the ADC configuration information for the particular device, in the form: AdcSubsys,AdcConfigure -> AdcDeviceP.

It effectively performs a callback into the component to obtain the necessary information.

Figure 53: ADC Configuration Callback Mechanism

1.4 Joystick

The RF04EB is equipped with a digital joystick, however it is wired through a potential divider to reduce its pin count, as shown in Figure 54 requiring the use of the ADC in order to determine its direction. It is also connected to one of the CC2430’s GPIO pins, which allows joystick movement to be detected through the generation of an interrupt request.

Figure 54: Schematic of the Joystick: Wired Through a Potential Divider to Reduce Pin Count [121]
The design, shown in Figure 55, is very similar to the potentiometer as described in Appendix E: Section 1.3 above, and the position of the JoystickP component in the diagram should be treated in the same way, in that Figure 53 is a truer representation of the component inter-relationship levels.

Figure 55: Design of the Joystick Abstraction
Appendix E - RF04EB Sensorboard Circuit Diagrams
Appendix F - MARCO Sensorboard

This is a custom sensorboard specified and designed by the author\(^2\), shown in Figure 56. The board’s design allows experimentation in a number of application classes and research areas to be performed. This was achieved by purposely bringing the pins from the MARCO robot’s edge connector and the CC2430EM evaluation module to pin headers which can be wire-wrapped for the required configuration. The circuit diagram of the board design is provided in Appendix G.

![Figure 56: MARCO Sensorboard: Shown Connected to MARCO Robot](image)

Further, the board can be used standalone, allowing daughter boards to be used without the RF04EB, or can be used with the MARCO Robot (Microprocessor Application for Robotic Control) [123] developed by the University of the West of England, Bristol.

The MARCO sensorboard provides an RS-232 port (can be switched to allow the use of the pins for GPIO) and 3 LEDs, in addition to exposing the pins of the CC2430EM evaluation module. The LEDs are connected to the same pins of the CC2430 as the RF04EB, as TinyOS assumes all mote platforms will provide three LEDs at a minimum, therefore considering the CC2430EM does not have any LEDs a decision was made to pass the assumption on to the sensorboard.

The MARCO robot is equipped with:

- A pair of DC motors to provide steering, fitted with shaft encoders for accurate positioning.
- A stepper motor to allow a sensor to scan through 180 degrees - no sensor fitted.
- A pair of micro-switches to detect the limits of the stepper motor rotation.
- Pairs of infra-red (IR) transmitters and receivers used for collision detection.
- A pair of reflective optical sensors to allow a white line to be followed.

1. Design & Implementation

It should be noted that the MARCO robot is not characteristic of a typical WSN sensorboard, however it does pose many of the same issues and provides another platform for application development with a different suite of inputs and outputs. The most significant difference is that energy efficiency is not a concern as it is powered by a laptop battery, however for the purposes of this project the considerations and implications of the abstraction designs on energy efficiency have been taken into consideration.

---

\(^2\) With thanks to Jim Whetstone for help with fabrication of the board.
The features and sensors provided by the MARCO sensorboard are sensorboard-specific. They will only be used by applications which utilise this board and, as a result, provide a high-level custom interface that provides logical operations for the device, as opposed to a generic sensorboard-independent interface such as Get and Set which provide no advantage over using the GeneralIO HIL interface directly, nor do they express the device’s functionality, pushing the burden of control up to the client of the abstraction.

1.1 Motors & Pulse-Width-Modulation

The motors fitted to the MARCO robot are driven through the use of pulse width modulation (PWM). PWM allows for an effective voltage to be generated from the relative time that the signal is on, known as the duty-cycle (see Figure 57), it enables a variable voltage to be efficiently achieved without the need for digital-to-analogue converters (DACs); for example, if the duty-cycle is 50% the effective voltage is half the maximum voltage. Some of the CC2430’s timers are capable of acting as PWM controllers, generating varied duty-cycles, by utilising the interrupt on compare functionality that some of these have.

Implementing PWM in a manner that does not prevent concurrency in the system from occurring necessitates the use of timers. Specifically, interrupts generated by a timer allow the event to propagate through the application’s component hierarchy where they can be dealt with by the appropriate module. The timers on the CC2430 signal interrupts on two event types: timer overflows and comparator matches. Timer overflow cannot be usefully used for PWM as its period is fixed, only allowing a 50% duty-cycle, so comparators must be employed. These enable the timer to generate interrupts when it reaches a specified count, which can be used to create the desired duty-cycle for PWM.

The sleep timer is equipped with a single comparator which is dedicated to the millisecond timer required by TinyOS, preventing its use for PWM; it would be possible to use an instance of the virtualised millisecond timer, however because its operations run synchronously in task context, the precision of the timer is far less accurate than obtained through the use of a timer’s Alarm interface (which requires a dedicated comparator per alarm instance). The delayed response to the event is particularly significant for PWM as it requires a consistent duty-cycle to be generated; in the case of the motors an inaccurate duty-cycle would cause the robot to drive in a large arc rather than a straight line.

The period and size of the timer is immaterial, it is the duty cycle that controls the speed of the motors. For this reason the Alarm interface of Timer3 was exploited because of its two compare registers and its limited capabilities for duties requiring a counter (it is an 8-bit counter incremented at 250kHz, causing it to overflow approximately every millisecond). The two compare registers allow the left and right motor to be independently controlled with a single timer, which produces a smaller code footprint than using a separate timer for each motor; however this does have other implications on the motor abstraction’s design.
A number of implementation possibilities exist, with respect to timer and timer feature usage:

- The CC2430 provides hardware support for PWM through its timer’s comparators. The timers can be configured to count in numerous ways and to perform a number of actions on comparator matches, as shown in Figure 58, however the output pin which exhibits this action is limited to one of two locations (Figure 59).

- The interrupt-on-compare function of the timer can be utilised to setup a duty-cycle, and handle the signalled events in software, setting and clearing the pins as required.
  - This can be achieved through the use of a single comparator that is changed each time the interrupt occurs in order to generate the desired duty-cycle; this enables a single timer to provide two PWM channels.
  - Alternatively, two comparators can be used to generate the duty-cycle, but this results in the timer only providing a single PWM channel.

Although the first option has slightly more computationally efficiency, as once the timer configuration is set the PWM control can be performed completely in hardware, when the implications of the trigger edge on GPIO interrupts for the CC2430 are considered (discussed in Section 11.9.1) this technique can be deemed unsuitable as it potentially prevents the CC2430 device being suitable for an application. Additionally, there are scenarios where using a single timer for independent control of the left and right motor is impossible with this method; two separate timers could be used to combat this, comprising code size for runtime performance.

As a result of the above implications the second method that uses a single timer was chosen for the implementation of the MotorPwmC component, which satisfies the requirements of the motor abstraction. However, this is another example where multiple implementations which target different objectives can be beneficial and highlights the challenge and level of understanding required in architecting a system; the use of the placeholder pattern for PWM implementation selection would allow this to be elegantly achieved.
### T3CCTL0 (0xCC) – Timer 3 Channel 0 Compare Control

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Reset</th>
<th>R/W</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>–</td>
<td>0</td>
<td>R0</td>
<td>Unused</td>
</tr>
</tbody>
</table>
| 6   | IM    | 1     | R/W | Channel 0 interrupt mask
0: interrupt is disabled
1: interrupt is enabled   |
| 5:3 | CMP[2:0] | 000 | R/W | Channel 0 compare output mode select. Specified action on output when timer value equals compare value in T3CC0
000: Set output on compare
001: Clear output on compare
010: Toggle output on compare
011: Set output on compare-up, clear on 0 (clear on compare-down in updown mode)
100: Clear output on compare-up, set on 0 (set on compare-down in updown mode)
101: Set output on compare, clear on 0xFF
110: Clear output on compare, set on 0xFF
111: Not used |
| 2   | MODE | 0     | R/W | Mode: Select Timer 3 channel 0 compare mode
0: Compare disabled
1: Compare enable |
| 1:0 | –    | 0     | R/W | Reserved: Set to 00. |

Figure 58: Timer3 Compare Register for Channel 0, showing Compare Output Action.

Note: The Timer3 Channel 1 Comparator Provides Identical Functionality

#### Table 56: Peripheral I/O Pin Mapping

<table>
<thead>
<tr>
<th>Periphery / Function</th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>A7</td>
<td>A6</td>
<td>A5</td>
</tr>
<tr>
<td>USART0 SPI</td>
<td>C</td>
<td>SS</td>
<td>MI</td>
</tr>
<tr>
<td>USART2 UART</td>
<td>RT</td>
<td>CT</td>
<td>TX</td>
</tr>
<tr>
<td>USART2 SPI</td>
<td>MI</td>
<td>MO</td>
<td>CS</td>
</tr>
<tr>
<td>USART1 UART</td>
<td>RX</td>
<td>TX</td>
<td>RT</td>
</tr>
<tr>
<td>TIMER1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>TIMER3</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>TIMER4</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>32768 Hz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEBUG</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 59: Peripheral I/O Pin Mapping Options on the CC2430

The motor abstraction provides a choice of two implementations; one has basic drive capabilities, whilst the other provides full PWM control allowing the drive speed to be varied. These
implementations are mutually exclusive, requiring the application developer to choose which one satisfies the application’s needs. The design of the MARCO motor abstraction employed the placeholder design pattern [55] as shown in Figure 60, which allows the implementation of the service to be easily changed at the application level whilst ensuring that only one implementation is utilised.

**Figure 60: The Placeholder Design Pattern [55]**

The realised implementation of the motor placeholder configuration is presented in Code Sample 58. The implementations provide the same interface, *MotorDrive*, so that an application can use either, as shown in Figure 61.

```
configuration MotorsC
{
    provides interface MotorDrive as LeftMotor;
    provides interface MotorDrive as RightMotor;
    provides interface Init;
    uses interface MotorDrive as LeftMotorImpl;
    uses interface MotorDrive as RightMotorImpl;
    uses interface Init as InitImpl;
}
implementation
{
    //just forward the interfaces
    LeftMotor = LeftMotorImpl;
    RightMotor = RightMotorImpl;
    Init = InitImpl;
}
```

**Code Sample 58: Placeholder Configuration for the MARCO Motor Abstraction: MotorsC**

The onus of implementation selection is placed on the application developer, requiring them to wire the *MotorC* abstraction to the desired implementation (Code Sample 59), but it is only the developer who knows what motor capabilities are required for their particular application. If an application uses the motor abstraction but forgets to wire to an implementation a compilation error will be displayed alerting the developer of their mistake, which prevents misuse. By providing a choice of implementations the generated executable size will be no larger than absolutely necessary; for example if only basic motor control is required the *MotorBasicP* module can be used preventing the inclusion of all components for Timer3 related to PWM control.

```
configuration MotorTestAppC
{
}
implementation
{
    components MotorsC, MotorsPwmC;
    //wire the placeholder to the desired PWM motor control implementation
    MotorsC.LeftMotorImpl -> MotorsPwmC.Left;
    MotorsC.RightMotorImpl -> MotorsPwmC.Right;
    MotorsC.InitImpl -> MotorsPwmC.Init;
    ...
}
```

**Code Sample 59: Example Usage of the MotorsC Abstraction**

The CC2430 is connected to the motors by four GPIO pins: two control the speed of the motor and the other 2 control the rotation direction of the motor. The *MotorDrive* interface presents logical directions with respect to the MARCO robot, such as forward and backward, which masks the actual rotation direction of the motors; this prevents the developer from forgetting that the motors must rotate in opposite direction in order to drive the robot in a straight line.
The duty-cycle parameter is ignored by the MotorsBasicP implementation.

Figure 61: MARCO Motor Design Incorporating a Basic & Advanced Interface
1.2 Position Encoders / Motor Position

The interface operations for the motor’s position and the design of the abstraction are presented in Figure 62; there are a number of points in this design that deserve discussion.

The position encoder employs a sensor which varies its output depending on the color of the wheel segment it is detecting, as depicted in Figure 63. The `moved()` event is triggered by a GPIO interrupt on a rising edge. While it is possible to increase the accuracy of the motor position through the generation of both rising and falling edges interrupts, this imposes restrictions on which pins the position encoder input can be connected to, and potentially causes complications on the use of other sensors, as discussed in Section 11.9.1. Considering the maximum positional error (illustrated in Figure 63) with the standard MARCO wheels is approximately ±2mm the decision to generate interrupts on a single edge-trigger is sufficient for practically all applications requiring this facility.

The encoder count is maintained by a non-volatile member variable with component scope (i.e. it is on the heap), which is incremented, or decremented depending on the motor’s rotation direction (obtained from the `GeneralIO` motor direction interface for the specific motor), every time the interrupt is fired.

The interface’s `getEncoderCount()` operation returns a count of the number of segments that have rotated past the sensor since the last time the counter was reset. The returned value will be positive if the motor has travelled forward, with respect to the front of the robot, and negative if the motor has travelled backward; the returned count does not indicate whether the motor has travelled in both directions, just its relative position from the last time the counter was reset. For this reason the `moved()` event is provided by the `MotorPosition` interface which allows the client of this abstraction to capture every movement of the motors that is performed.

Calculating the distance travelled is left to the client of this abstraction, as this functionality would be seldom needed and reduce runtime performance; especially if floating point arithmetic was necessary.

Figure 62: Design of the Motor Position Abstraction & Interface Definition
1.3 Bumpers

The design of the bumpers abstraction, shown in Figure 64, is simple. The configuration, BumpersC, wires the interfaces used by BumpersP to the appropriate GeneralIO and GpioInterrupt interfaces for the pin that the motor is physically connected to.

The Bumpers interface provides operations to enable and disable interrupt generation when this functionality is not required, which prevents the CC2430 from being woken from sleep mode when this is not necessary.

This design is sufficient assuming the motors bumper sensors are always connected to the same GPIO pins on the CC2430. If this is not the case however a simple modification to BumpersC which utilises pass-through wiring defers the decision of which GPIO pin to use to the abstraction’s user, who must then declare the nesC component wiring as shown in Figure 65.
The interface provided by the line sensors is slightly more complex than that provided by the bumpers, to pass responsibility of wiring the nesC components together to the user of the abstraction.

Figure 65: Alternative Abstraction which Passes Responsibility of Wiring to the Client

### 1.4 Line Sensors

The line sensor design is similar to the bumpers and can use the same method described for the bumpers to pass responsibility of wiring the nesC components together to the user of the abstraction.

The interface provided by the line sensors is slightly more complex than that provided by the bumpers, in order to give an application using the line sensors the flexibility of implementing a line or edge-of-line follower for example. The user must be aware of the implications of changing between `interruptOnLineFound()` and `interruptOnLineLost()`, as they modify the edge-trigger for the interrupt (implications discussed in Section 11.9.1).

Figure 66: Bumper Abstraction Design & Interface Definition
Appendix G - MARCO Sensorboard Circuit Diagrams
Appendix H – Paper & Reviewers’ Comments: Shaping TinyOS to Deal with Evolving Device Architectures: Experiences Porting TinyOS-2.0 to the Chipcon CC2430
Appendix I – Initial Project Plan: Gantt Chart
References


[84] Armega128 GPIO configuration available from:


[85] Generic Armega128 GPIO pins module available from:


