

PIC-based TinyOS Implementation

Ciarán Lynch and Fergus O' Reilly

Centre for Adaptive Wireless Systems

Cork Institute of Technology

Cork

IRELAND

Email: ciaranlynch@cit.ie, foreilly@cit.ie

Abstract—Management of wireless sensor networks is an area of research which aims to minimise the power dissipation of such networks while maximising their functionality. TinyOS is a widely used operating system for modern wireless sensors, as its modular system is flexible yet powerful. In order to port TinyOS to a new platform, the nesC compiler must be adapted and some operating system modules must be ported. We describe the issues involved in running TinyOS on a PIC-based wireless sensor, and how the compiler must be modified to cope with the limited PIC architecture. We show that the final performance is comparable to or better than existing sensor systems for a limited set of applications, but that complex applications are likely to prove extremely difficult to implement.

I. INTRODUCTION

Networked systems of miniature sensors are being hailed as a revolutionary development, which will change the way people interact with technology [1]. Two initial deployments of such systems are simple environmental monitoring applications such as wildlife monitoring [2], where a network of sensor nodes was used to monitor temperature and sunlight in a remote bird sanctuary without interfering with the nesting species there, and intrusion detection [3], in which concealed nodes were deployed in a military application as a virtual trip-wire which detected any object crossing a certain line and used a distributed algorithm to classify the target as human or vehicle by the vibrations generated, and then as military or civilian by the metallic content of the target. Both of these systems relayed their results back to a fixed network. Features such as distributed localization [4] and power harvesting [5] will expand the scope of these networks, eventually making possible the notion of true ubiquitous computing.

The software system considered here is based on earlier versions of TinyOS, developed at the University of California, Berkeley [6]. This uses nesC [7], a C++-

like object-oriented language which is compiled into C [8].

The development of sensor networks has extended the scope of the software system, and sensor networks in general, with advanced microcontrollers and more complex radio protocols such as Bluetooth [9] and Zigbee [10]. While these broaden the set of applications possible, this generally comes at the expense of power. We consider the opposite end of the scale – a very simple microcontroller and RF transceiver, and investigate the challenges of implementing TinyOS on such a limited system.

This paper discusses the implementation of TinyOS using nesC on a PIC-based sensor developed at the Cork Institute of Technology, Cork, Ireland (CIT). Due to the limitations of the PIC architecture compared to the more flexible AVR microcontroller used by the Mica sensor nodes, some modification of the code was required.

The hardware architecture of the sensor node developed at CIT is discussed in Section II. The general software architecture is discussed in Section III, the scheduler in Section IV, and the system compiler in Section V. The performance of the compiler and hardware is discussed in Section VI and compared with existing sensor platforms. Section VII concludes the paper.

II. HARDWARE ARCHITECTURE

The sensor node developed at CIT is represented in Fig. 1. It is controlled by a Microchip PIC16F877 microcontroller [11]. Previous work describes this sensor node in more detail [12].

The radio transceiver is a Nordic nRF903 868MHz FSK transceiver [13], operating at up to 76.8kbit/s. It operates as a pure transceiver, and as such is sensitive to timing jitter in the controlling processor.

III. SOFTWARE ARCHITECTURE

Most of the operating system code is written in nesC. This is an object-oriented extension to the “C” language

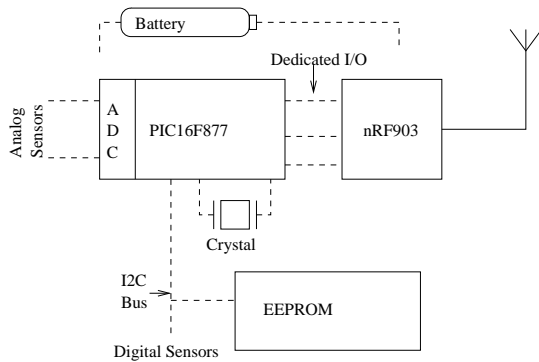


Fig. 1. Sensor module hardware

which allows code to be contained in modules with defined interfaces. The nesC language is discussed in detail in the nesC reference manual [14], [6].

The code is divided into modules – each module has a defined interface and other modules will only interact with it using methods from the interface. The interface definition is bi-directional, specifying which methods are called from external modules as well as which of its methods external modules may call. Each module may provide multiple interfaces, and may provide a configuration file which links it to other modules (allowing encapsulation of modules – for example the Timer code is contained in the configuration `TimerC`. This links to `ClockC` internally, which interfaces to the actual timer hardware but the application is isolated from this).

The module is implemented in nesC, which is compiled into C by the nesC compiler. The application running on the sensor is also a module, and is implemented in the same way as any other module. The structure of the PIC targeting nesC compiler is shown in Fig. 4, and discussed in Section V.

A. Radio

The biggest system represented is the radio system, since this is the most complex piece of hardware in the system. The Message layer takes a message and decides where it should be sent. This involves looking at the message type and the destination to determine whether the message should be sent over the radio or serial port. This level is not implemented in the PIC-based sensor. The `RadioPacket` module is responsible for encapsulating the message in the required packet format for the radio, and `SerialPacket` with that of the serial port. It passes the packet to the byte layer, which is responsible for sending the message bytes, encoded if necessary (for example some systems use Manchester coding to improve reliability) to the byte level system.

The UART operates at the byte level, so this is the hardware interface for that level and there is no bit-level module. The radio is serviced at the bit-level, so there is another level responsible for shifting bits out to the radio.

B. Timer

The timer system interfaces with the system clock. An external crystal is used to allow the processor to sleep and then wake up on a clock event, allowing for very low-power operation. It interfaces directly with the timer hardware – although it is quite high in the module graph for this, it is quite processor-specific anyway, for efficiency (timer events are generally the most frequently occurring, as events are scheduled for future execution and the processor sleeps, so they must execute as quickly as possible).

C. Logging

The data-logging interface is represented at the packet level – it takes sixteen bytes of data and writes it to the off-chip EEPROM using the I²C bus. The I²C bus master protocol is implemented in software at the bit level by the I2C module.

D. Sensors

The sensor interface is abstracted from the actual protocol used to communicate with the sensors. Modules interfacing with analog sensors include the ADC module internally, those using digital sensors include I2C or whichever other method they use. The application calls a method to read a value, which starts the conversion and then returns. When the conversion is complete, an interrupt is signalled which is translated into an event in the application.

E. System

The `RealMain` module provides the link from the operating system startup code to the application and the modules. It calls the `StdControl.init` method, which any module (or application) can link to if it needs to perform initialisation and a `StdControl.start` method which can be linked to to start up a module or application. In order to be started up in this manner, the application should include the module `RealMain` in its configuration and link `RealMain.StdControl` to its own `StdControl` interface.

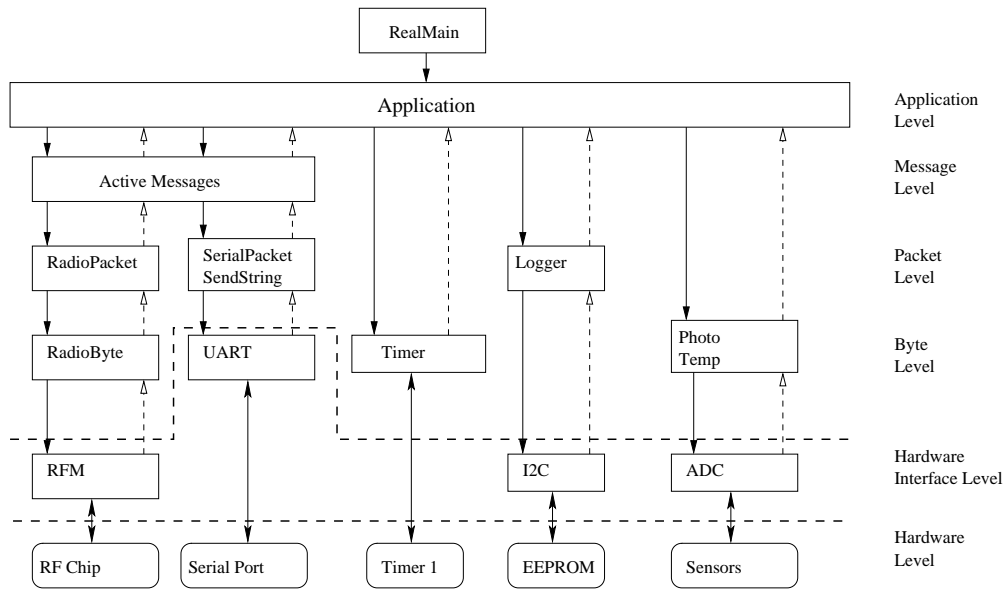


Fig. 2. Module graph.

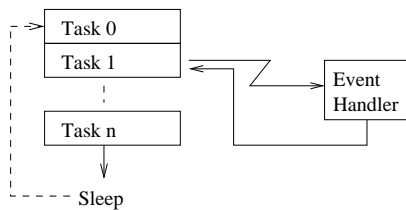


Fig. 3. Two level scheduler

F. Other

Not shown in the graph are some miscellaneous systems, which do not easily fit. Some of these deal with specific hardware, such as the Watchdog interface which interfaces with the watchdog timer, if enabled. The LEDS interface provides an interface to PORTB on the PIC, or the debugging LEDS on the Berkeley sensor nodes.

IV. SCHEDULING

Due to the resource constraints of the networked sensor, the more common scheduling algorithms used in larger systems are not appropriate and pre-emptive scheduling is impossible. A very simple, two-level scheduler is used, as shown in Fig. 3. There are two types of process – tasks and events.

A task is the basic processing unit. Tasks may be scheduled by the application or by the operating system, and are executed in a FIFO queue. Tasks may not preempt other tasks. Computation that runs for a long time should be broken up into multiple tasks, to allow any other tasks that have been scheduled to run. The

queue is of a fixed length and if too many tasks are posted, attempting to post additional tasks will fail.

Events generally represent notification of an asynchronous hardware event. Events may pre-empt tasks, but not other events. For this reason, events should execute as quickly as possible, or the system could miss some other pending events. Generally an event handler should only change some status flags or call some commands – any prolonged processing should be passed off to a task. Reading bits from the radio uses a timer event handler so if another event handler runs for too long radio transmission may be corrupted.

When there are no tasks scheduled to run, the system goes into a low power sleep mode until woken up by an external interrupt or the asynchronous timer. In most sensor applications the system will be asleep for up to 99% of the time, prolonging battery life.

This represents ideal operation of the scheduler. In practice, this performance is not always possible on the PIC-based system, for reasons discussed in Section V

V. COMPILER IMPLEMENTATION

The nesC compiler links all of the required modules together and creates a single C file containing all of the code. Starting at the application module, it scans the list of modules included. Each module has a private data area, and its own private functions as well as the public interfaces it provides (similar to a C++ class, except that a module can only ever be instantiated once in each application) – all of the function calls and data references

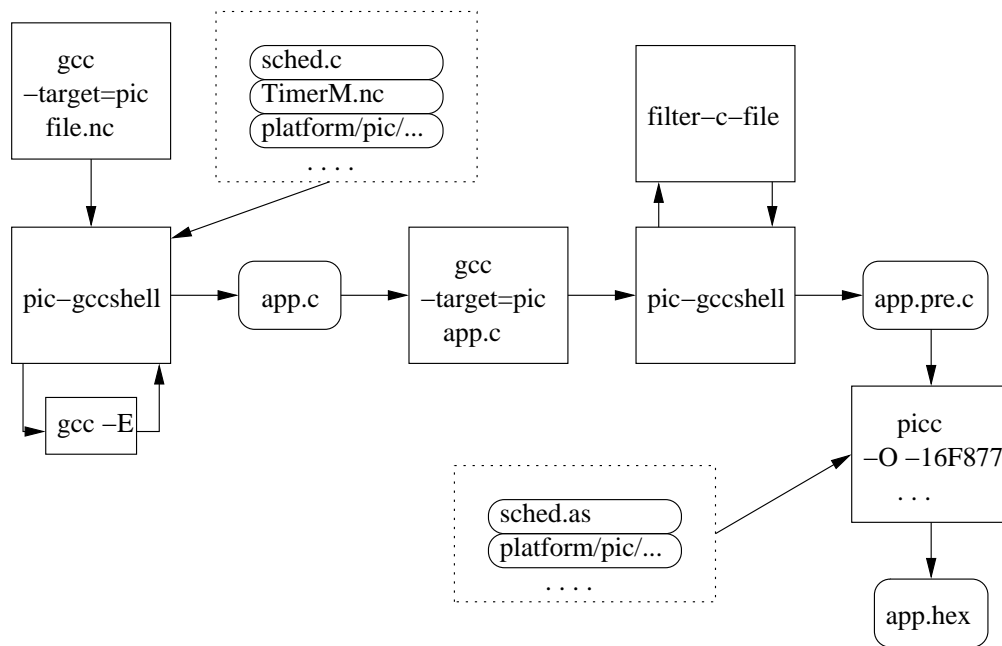


Fig. 4. PIC nesC compiler

are converted to a fully-qualified name, containing the module name and the function or variable name. The bulk of the module code is essentially unchanged, as the nesC syntax closely follows that of ANSI C.

In the case where a method call results in more than one function being called, the results are automatically combined. The definitions are output to a file, followed by the entire C code for each module.

Adding the PIC as a new target to nesC requires only modifying a single file and recompiling, however the C code produced by nesC is not suitable for compilation by the PIC C compiler (the HITECH PICC compiler [15]), and some filtering was required on the code and on the compiler itself.

The filter scripts were written in PERL, a general-purpose scripting language with very powerful text manipulation and pattern matching libraries.

A. Compiler Invocation

A wrapper script was written to translate the options given to the C compiler. nesC invokes the compiler directly, and while it can be made to call the PIC compiler when the PIC target is used, it assumes a gcc-like syntax (the Atmel C compiler is a port of gcc). The PIC C compiler has its own set of options, which must be translated.

The preprocessor used in the PIC C compiler is also quite limited – it does not support identifiers more than 31 characters in length, macros with variable argument

lists or complex constructions of macros (it conforms to the minimum standard but is quite restrictive for a modern C compiler). For this reason, the wrapper determines whether the file is being preprocessed or compiled and uses gcc to preprocess and PICC to compile. This is handled by the script `pic-gccshell`, as shown in Fig. 4.

B. Non-standard Code

The PICC compiler, as with almost every embedded C compiler, extends the language to allow better resource utilisation. The PIC data RAM is divided into four banks, the `bank0`, `bank1` up to `bank3` keywords allow placement of variables in a specific bank. There is also a directive to place a variable at an absolute address, and to define bit-variables. The gcc pre-compiler will not pass this code (since it is not standard C). Another filter script was written which is run on the C file that is output by the nesC compiler. It adds in the special-function register definitions at their absolute addresses. It also translates declarations of a specific type to use the `bankn` keywords – a declaration of the form `typedef xxx_t xxx_bankn_t`; is transformed to `typedef bankn xxx_t xxx_bankn_t`; – allowing the use of these features without interfering with existing code.

While this will work well for code written for this compiler, it does not aid existing code, which will by default place all data in bank zero, and not use three quarters of the available memory. The compiler therefore

scans through the code and moves any data declaration that is not already allocated a bank. Function auto variables are left in bank zero, system data such as the task list is placed in bank one, radio buffers are placed in bank two and all other global variables are placed in bank three.

Although an even distribution between banks would be desirable, it would cause problems with pointers, since the PIC C compiler requires that code know which bank a pointer is accessing. The PIC only uses a seven-bit pointer type. This stores the offset within the bank but the specific bank used must be specified. (for example a `bank1 char *` and `bank2 char *` could have the same value but refer to different locations) Separating the data into banks by data types allows this to be taken care of in the data types.

gcc also uses some constructions which are not ANSI C such as inline functions – these must be removed from the code using the filter script. The Atmel gcc port has its own syntax for specifying which functions are used as interrupt handlers – this is replaced with that of the PIC C compiler. All of this filtering is performed by the script `filter-c-file`, shown in Fig. 4, which is called from the `pic-gccshell` script.

C. Compiled Data Stack

The PIC architecture does not easily support a data stack, as would be used for most C implementations. The PIC C compiler uses a compiled stack. Static resources are allocated for each function using a call graph. Since the storage space for the function arguments is also static, simply calling a function will overwrite the arguments even if no local variables are used. Recursive function calls are clearly not possible, and no function may be reentrant.

In order to guarantee this, each task either disables interrupts for its critical sections. The compiler constructs a call graph and a list of functions called by the interrupt handler. Interrupts are then disabled before calling any functions in this list from static code.

This introduces some extra overhead to the system, this is discussed in Section VI.

D. Optimisation

The changes described above are enough to have the code compile for the PIC, however the generated code is very inefficient. nesC assumes that gcc is being used for the final compilation. gcc supports inline functions and can be made to optimise code quite aggressively, so the code generated by nesC need not be very efficient.

In particular, it makes heavy use of function calls. Every module call is translated to a function call, and if this links to a sub-module it is called as another function.

A function is also used to combine the results when there is more than one function linked to a particular call. The PIC only supports an eight-level call stack (including interrupt handlers) so a single call occurring from an interrupt, while another call is active may be enough to overflow the call stack.

For this reason, the filter script must perform optimisation, particularly to reduce the depth of the function calls generated by module wiring. It identifies functions that do nothing but pass their arguments on to other functions, and replaces them with a macro, which calls the appropriate functions. This is carried out using PERL's powerful regular-expression matching features. The module wiring code is automatically generated by the nesC compiler and conforms to a strict syntax. Passing the code through multiple preprocessors tends to introduce a lot of "line-number" directives to allow the compiler report error messages in the appropriate place in the original code file but the regular expression engine is powerful enough to filter these out.

Functions which only return a constant or which evaluate to a simple expression are replaced with this expression. The function result combining is also replaced by a macro.

In this case, the functions are identified and removed by the PERL regular-expression matching, and their definition is replaced by a macro. The code is then passed through the preprocessor again to place these macros into the code. This is done quite efficiently by forking another process which runs gcc, and then piping the output of the filter script directly into gcc.

Although this might seem to increase code size since the code will be replicated every time it is used, it is a very simple logical operation and actually ends up being smaller than the code to call the function which implements it. Similarly, the code to enter and exit an atomic section is defined as a function, but is replaced by a macro which is smaller than the function call overhead, and saves one level of call stack. The `filter-c-file` script carries out all of this optimisation.

Both the gcc preprocessor and PERL are used in different places to carry out the complex matching and replacing needed to perform this task. PERL has an extremely powerful pattern-matching and processing engine, but it operates on the source file as a large text file, with no regard for the functionality of the code contained in it. The gcc preprocessor is much simpler

and is only capable of simple macro operations but this is carried at the token-processing level, so it can modify the syntax of the actual code.

E. Scheduler Implementation

The scheduler is implemented using standard C in the original Atmel TinyOS implementation – a task is scheduled by passing a pointer to the task function, which is then executed by calling the function pointer. On the PIC this is quite a costly operation, as it requires three levels of the call stack to call a function using a function pointer. For this reason, some custom assembly code is used, along with a preprocessor macro, which takes the function address directly and loads it into the task queue, and then loads the Program Counter with this address to jump to it. As mentioned before, the function call stack on the PIC is fixed at eight entries so it is very important to limit its use, particularly inside an event handler. Due to their asynchronous nature almost all events are triggered from inside the interrupt handler. The limited call stack may require disabling interrupts for some of the operating time, this is discussed in more detail in Section VI.

VI. RESOURCE USAGE

The PIC-based sensor platform is extremely resource-limited, so the operating system must be as small as possible. Typically the sensing applications will be required to run unattended for periods ranging from months to years, so power consumption must be kept to an absolute minimum. Due to the limitations of the PIC, some code must be modified and some speed sacrificed to allow the code to execute correctly.

A. Call Depth

Minimising function call depth decreases interrupt latency – as discussed in Section V, interrupts must be disabled whenever the function call depth could exceed eight if an interrupt occurred.

CntToLedsAndRfm is an application that is part of the TinyOS distribution. It runs a counter, displays the value on the LEDs and transmits it over the radio. When compiled for the PIC with no optimisation, it has a task function call depth of eight (which means interrupts must be constantly disabled) and an interrupt depth of nine (which will not execute) – enabling these optimisations reduced these to six levels in tasks and six in the interrupt handler.

The overhead this introduces in practice is discussed in the next section.

B. Locking Overhead

As discussed in Section V, interrupts are disabled whenever the execution of an interrupt could result in the call stack overflowing. Since the PERL-based filter script is not aware of the complex C syntax, it can not insert code directly into a function (the point at which the call depth is exceeded can occur in user-generated code which can take any form, unlike in module wiring optimisation where only system-generated code is involved which conforms to a strict syntax). If a function X is the first level at which interrupts must be disabled, the system replaces the function name where it is declared with REAL_X. It then inserts its own function named X which disables interrupts and then calls REAL_X.

The overhead of locking is therefore to reduce by one the effective size of the call stack. It also requires interrupts to be disabled for potentially quite long periods, which increases interrupt latency substantially. This can result in inaccurate timers, or radio packets being lost if an event can not be handled in time. In the worst case, when the interrupt depth is six or greater, interrupts are disabled whenever a task is running, effectively reducing the scheduler to a single-level scheduler as an event can no longer preempt a task.

C. Power

The hardware platform is concurrently in development, but some preliminary results can be predicted from the datasheets of the various components. The results obtained for power dissipation are shown in Table I. The three components that dominate power dissipation are the processor, the radio and the EEPROM. Usage of the radio and EEPROM is extremely application-specific, so it is difficult to predict a ‘typical’ usage.

Assuming no radio or EEPROM activity, the idle power is $26\mu\text{W}$, and the active power is 1.525mW . Assuming a 1% duty cycle, a typical goal for sensor applications, the average power dissipation is $41\mu\text{W}$. Using a standard Energizer CR2450 575mAh, 3V miniature lithium battery gives an expected battery life of approximately 4.8 years, assuming an ideal battery with no loss or time-related discharge.

This assumes no radio transmission. The radio transmits at 20kHz, so each bit transmitted requires approximately $1.2\mu\text{J}$ of energy (0.101nAh per bit). If the radio is used significantly, this should be factored into power calculations. Transmitting one 29-byte packet (at 20kbps) per minute will bring the average power dissipation up to $47.4\mu\text{J}$, and the battery life down to 4.1 years. Listening is almost as expensive as transmitting (power dissipation

TABLE I
PREDICTED POWER DISSIPATION

Component	Idle	Active (Tx/Rx)
Processor	1 μ W	1.5mW
Radio	24 μ W	33/24 mW
EEPROM	1 μ W	15/2 mW

is dominated by the mixers and filters which are active for both receive and transmit) – so listening for one packet interval per minute will bring the battery life down by almost as much (to approximately 3.5 years). For this reason, low-power listening strategies must be implemented, or power must be obtained from another source. A typical home or office environment can contain enough vibration energy for a 1cm³ piezoelectric converter to pick up between 50 and 100 μ W [5], although not all deployments will allow this.

This compares well with the early Berkeley sensor nodes, which are of similar complexity. Measured values from the Rene motes [16] (from 2000) give an average power dissipation of over 200 μ W and an expected battery life of less than one year. Later designs improve on this by using application-specific digital logic, at the expense of increasing system cost by moving away from off-the-shelf components.

D. Code Size

Direct code comparisons are difficult, due to differences in architecture, but approximate comparisons may be made. The PIC uses a 14-bit ROM word to store program instructions; the AVR processor used in the Rene mote uses two 8-bit bytes per instruction. The results given are in bytes, assuming that one PIC word is equivalent to two AVR bytes. Both architectures use 8-bit bytes for data memory, however the PIC statically allocates RAM for the entire program data memory (including function auto variables), while the AVR only stores static and global variables statically, and allocates function auto variables from a data stack. This is not reserved and its size may vary according to how the functions are called, but must be available at runtime. Code and data sizes for a simple application, which flashes an LED at a frequency of 2Hz, are given in the first two rows of Table II. The two values are almost the same, as expected since the two architectures are quite similar.

The final two values are for an application which runs a counter at 1Hz and transmits a packet containing the

TABLE II
PROGRAM IMAGE SIZES

Application	Code(b)	Data(b)
Blink (ATMEL)	1428	44
Blink (PIC nesC)	1434	64
CntToRfm (ATMEL)	9156	353
CntToRfm (PIC nesC)	3324	115

counter value every second. This gives an idea of the size of the radio system, since as noted above, the size of the timer code in nesC is essentially the same for the two architectures. The PIC version is only 36% of the size of the AVR version. However, this value is qualified by the fact that the radio controller on the Berkeley mote is significantly more complex, and only needs to be accessed once per byte transmitted, allowing lower-power operation and freeing up more processor time on the microcontroller. The PIC radio controller is quite simple, and must be controlled at the bit level. This requires exclusive use of the microcontroller, and is quite power-intensive.

VII. CONCLUSION

TinyOS has demonstrated its suitability for sensor network management applications, however almost all of the work has focused on the Mica and related hardware platforms. In this paper, we have considered an implementation of TinyOS on a wireless sensor based on the Microchip PIC microcontroller. We have shown that the expected power dissipation for some typical simple tasks is lower for the PIC sensor than the Mica. However, the PIC does not allow the same amount of flexibility as the Mica, and this lower power dissipation must be traded off against greater interrupt latency and restrictions on the complexity of the code. The maximum code size for the Mica is substantially higher, but again most practical sensing applications do not require very large code images. Optimal use of the PIC resources requires more care than the Mica, as data memory banks must be considered, and care must be taken with function call depths. However, for a small increase in design time, considerably lower power dissipations are possible with little decrease in performance, in a simpler and cheaper system. Advanced processing and networking are likely to be extremely difficult to implement on the PIC, although future work will investigate newer and more flexible PIC architectures which should allow

more advanced applications with little increase in power dissipation.

REFERENCES

- [1] Wired Magazine, "Intel's tiny hope for the future," Dec. 2003.
- [2] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proc. ACM International Workshop on Wireless Sensor Networks and Applications*, Sept. 2002, pp. 88–97.
- [3] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita, "A line in the sand: A wireless sensor network for target detection, classification, and tracking," *Computer Networks*, vol. 46, pp. 605–634, July 2004.
- [4] D. Niculescu and B. Nath, "Ad hoc positioning system (APS) using AoA," in *Proc. 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 22, no. 1, Mar. 2003, pp. 1734–1743.
- [5] S. Roundy, P. K. Wright, and J. Rabaey, "A study of low level vibrations as a power source for wireless sensor nodes," *Computer Communications*, vol. 26, no. 11, pp. 1131–1144, 2003.
- [6] J. H. et al., "System architecture directions for networked sensors," in *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000, pp. 93–104.
- [7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," in *Proc. ACM SIGPLAN 2003 conference on Programming language design and implementation*, June 2003, pp. 1–11.
- [8] J. L. Hill, "System architecture for wireless sensor networks," Ph.D. dissertation, University of California, Berkeley, 2003.
- [9] *Bluetooth Specification v1.1*, Bluetooth Special Interest Group, Feb. 2001.
- [10] *IEEE802.15.4 Proposed Standard (Zigbee)*, IEEE 802.15 WPAN Task Group 4.
- [11] *PIC16F877 Datasheet, Revision C*, Microchip Technology Inc., 2000.
- [12] C. Lynch and F. O. Reilly, "Pic-based sensor operating system," in *Proc. Irish Signals and Systems Conference, Queen's University, Belfast, 2004*, June 2004, pp. 95–100.
- [13] *Nordic nRF903 Datasheet, Revision 3.1*, Nordic VLSI ASA, Dec. 2002.
- [14] D. Gay, P. Levis, D. Culler, and E. Brewer, *nesC Language Reference Manual*, Sept. 2002.
- [15] HI-TECH Software, *PICC Manual*, July 2003.
- [16] J. McLurkin, "Algorithms for distributed sensor networks," Master's thesis, University of California, Berkeley, Dec. 1999.