

# Processor Choice For Wireless Sensor Networks

Ciarán Lynch  
Centre for Adaptive Wireless Systems  
Cork Institute of Technology  
Cork  
Ireland  
ciaranlynch@cit.ie

Fergus O' Reilly  
Centre for Adaptive Wireless Systems  
Cork Institute of Technology  
Cork  
Ireland  
foreilly@cit.ie

## ABSTRACT

Networking and power management of wireless sensor networks is an important area of current research. The choice of microcontroller to control such devices is critical to their performance. We discuss some of the features that affect the performance of a microcontroller in sensing applications, with particular reference to power. Consideration is given to the particular demands of TinyOS, the most prevalent sensor operating system. We survey some popular existing devices, and suggest how the performance of a new design might be optimised.

## 1. INTRODUCTION

Wireless sensor networking is one of the most exciting technologies to emerge in recent years. Advances in miniaturisation and MEMS-based sensing technologies offer increases by orders of magnitude in the integration of electronic networks into everyday applications. Traditional microcontroller design strategies have not reached the best possible power consumption, especially for the specialised application set of sensing networks.

Power efficiency is a prime concern in wireless sensors, whether powered by a battery or an energy-scavenging module. Trends in miniaturisation suggest that the size of wireless sensors will continue to drop, however there has not been a corresponding drop in battery sizes. For example, a standard 3V CR2450 lithium coin cell has an energy density of 240mAh/cm<sup>3</sup> at 3V [4]. – a sensing application requiring 4mAh per day with a twelve month deployment would require 6.1cm<sup>3</sup> of battery. In one test deployment of the Mica2 motes, an environmental monitoring application had a daily energy budget of 8.14mAh [8].

In this paper we consider the microcontroller and its effect on the power consumption of the sensor node. We review the factors influencing the power consumption and calculate the expected performance of some existing hardware in sensor applications.

Section 2 describes TinyOS, Section 3 describes some of the significant hardware factors affecting the performance of the microcontroller. Section 4 discusses existing microcontroller hardware, Section 5 the power used by these devices and Section 6 concludes the paper.

## 2. TINYOS

TinyOS, originally developed at the University of California, Berkeley [6], has emerged as the leading operating system in research relating to control of wireless sensor networks. Its modularity and C-based syntax aim to provide a shallow learning curve for an experienced programmer.

All of the controlling software runs on the microcontroller contained in the wireless sensor node. TinyOS was originally designed for AVR-based microcontrollers. Recently, a port for the Texas Instruments MSP430 has been carried out, and previous work [7] demonstrated how the basic operating system and a subset of the libraries can even be written for such a limited device as the Microchip PIC16.

TinyOS uses a two-level scheduler – tasks run in a FIFO queue, with the system sleeping when no task is scheduled. Events run asynchronously, usually from interrupt handlers, and will interrupt any task that is running, or wake the system from sleep. It also includes a wide range of library modules.

## 3. HARDWARE

Power conservation is almost always the principle factor in TinyOS application design. Some of the factors affecting the power conservation of a particular device are considered here.

### 3.1 Wakeup Time

A widely used goal for sensor applications is an average duty cycle of 1%. This is achieved by scheduling events for some time in the future, and then sleeping. Most microcontrollers implement an asynchronous timer which is clocked while the core and other peripherals are powered off. An interrupt is triggered when this timer rolls over to wake the system from sleep.

Interfacing with external hardware often results in waiting times in the core. A typical sequence consists of setting some bits, waiting a number of milliseconds, changing the settings, waiting some more milliseconds and so on. If the wakeup is sufficiently fast, the core may sleep while waiting.

The primary factor determining this is how quickly the main oscillator can start up and settle into a stable cycle. Depending on how timing-sensitive the application is, it may not be necessary for the cycle to have completely settled. It may be useful to start execution before the oscillator has completely settled, and provide a flag indicating when it has settled to

within a certain tolerance of the nominal value which can be checked before particularly sensitive operations.

Some architectures may allow the oscillator to keep running but not be connected to the microcontroller core, allowing the core to start up within a few clocks of an interrupt. For long periods of sleep the extra power drawn by the oscillator is likely to dominate the system power dissipation but if it is known in advance that the sleep is short (for example from a timer) the fast wakeup time may allow it to sleep where it would not otherwise be possible.

Recently, some architectures have been proposed which use entirely asynchronous logic [3] eliminating oscillator startup delay. However these architectures are still experimental, and some work is required before functioning silicon will be available.

### 3.2 Clock Scaling

Some microcontrollers can dynamically switch their operating frequency, either by using a divider on the primary system oscillator or a low-frequency oscillator with a controllable multiplier. The first approach runs the oscillator at full speed, dissipating maximum power, but the system logic operates at a lower speed and lower power. The multiplier has its own power overhead, and could require time to settle if the multiplying factor is changed, whereas the divider will be able to switch frequencies instantly. The multiplier-based approach saves power by running a low-speed oscillator. There is usually a 32kHz oscillator for timing purposes, so there is no power overhead apart from that of the multiplier itself.

It is important to consider the overhead of the switch – depending on how the clock division is carried out it may require a number of cycles to settle before execution continues. This may result in “dead time” where the microcontroller is powered on but not executing anything, wasting power.

### 3.3 Memory Architecture

When servicing an interrupt, the handler will often have to save a number of registers and flags before executing anything. Since an interrupt handler will generally execute very little code, this can also come to dominate the time for which the core is active.

In the case of TinyOS, event handlers may not pre-empt each other. This means that the interrupt handler executes with interrupts disabled, and nested interrupts are not possible. It is therefore not necessary to store the values on a stack, since there will only ever be one stored state. The most efficient solution would be to have two completely separate sets of registers, and switch between them using a banked access, allowing the context switch to take place in a single instruction.

A compiled stack, as discussed in Section 3.6 separates the two stacks completely, however this may introduce overhead in other areas. It is addressed efficiently in any architecture, since the memory access is always a constant address. Accessing local variables on a dynamic stack is typically done using an indirect address of the form *base register + constant offset* [12] – this should be done in a single instruc-

tion. Protection of the stack from overflow and underflow will enhance the reliability of the application.

A segmented memory architecture, with registers, stack and static data all accessed differently can produce very efficient code, but may result in complications when using pointers as it is difficult to define a generic pointer.

### 3.4 I/O Issues

Since the microcontroller is required to control a number of other devices, it must have a large number of digital I/O pins available for general use. For example, the Nordic nRF2401 [11], a reasonably advanced modern radio transceiver operating at 2.4GHz, needs nine I/O pins connected to the microcontroller (six if only one data channel is being used). Digital sensors will generally have a serial interface, but this may still require up to four or five pins per sensor, as will an external FLASH memory.

Equally important is the logic behind them. Serial interface logic can allow the shifting out of digital data to be carried out in the background, or while the microcontroller core is powered down. Many digital sensors use standard serial bus protocols such as I<sup>2</sup>C or SPI, hardware support will allow these to be used more efficiently.

Almost all microcontrollers contain a hardware UART, which can be used to control a standard serial port, using an external level-shifting chip. Although this is far too power-hungry for a battery-powered device, it is common for some of the nodes to function as wired “base-stations”. The UART may also be used to control data clocking at the radio interface. Some UARTs may continue to run while the microcontroller core is asleep, waking it up with an interrupt whenever attention is required.

### 3.5 Instruction Set Complexity

The real power of sensor networks lies in the possibility of the network itself processing the data and taking action, or at least minimising the amount of data transmitted over the radio, saving power. Current microcontroller architectures generally do not lend themselves well to complex data processing tasks. The word size is eight bits, and general-purpose registers are usually scarce. Floating point calculations are inefficient.

The addition an *Add with carry* instruction makes carrying out sixteen or thirty-two bit arithmetic much easier on an eight-bit architecture. A hardware multiplier allows floating-point arithmetic to be carried out, although this will always be inefficient in an eight-bit architecture. More advanced instructions to carry out matrix operations, such as those found on some DSP processors allow digital filtering to be carried out at the sensor.

### 3.6 Software Issues

A processor with a large working set of registers will usually have a more flexible compiler than a processor with a load/store architecture.

Embedded compilers generally use one of two methods to assign function auto variables – a dynamic stack or a “compiled stack”. A dynamic stack is the traditional LIFO queue,

which is accessed by *push* or *pop* instructions. A compiled stack determines the data locations at compile time, using a function call graph to determine which locations can safely be overlapped.

The advantage of a dynamic stack is that any configuration of functions can be supported, and functions can (if desired) be reentrant. The disadvantages are that some embedded architectures may not have dedicated stack access instructions. In a compiled stack all memory accesses are static, since the address is determined at compile time, result in efficient memory access.

## 4. REVIEW OF EXISTING HARDWARE

The existing platforms with a full TinyOS implementation are the ATMEL AtMega128L, the AT90LS8535, and the Texas Instruments MSP430. Many other platforms are in development, or have partial implementations. These and some other common microcontrollers are compared under some of the headings discussed in Section 3.

The microcontrollers considered are the ATMEL ATMega128L [2], the Microchip PIC16F877 [9], the TI MSP430C1351 [13], the Analog Devices ADuC845 [1] which contains an 8051 core, and the Microchip PIC18F4525 [10]. This represents a wide selection of the microcontrollers currently available “off the shelf”. The MSP430 has a sixteen bit word size; all of the others use eight.

### 4.1 Power Saving

The MSP430 has six different power modes, ranging from fully active, to not clocking the core, to keeping the digital oscillator running to generate the clock but disabling the loop control to save power to fully powered down (with peripherals separately enabled or disabled). Due to the use of the digital oscillator, wakeup time can be as low as  $6\mu s$ .

The ATMega128L also has a variety of power down modes. The CPU clock can be stopped, leaving the peripheral clocks running. The CPU oscillator can be kept running – allowing it to restart in under one microsecond, or stopped. A special power-down mode stops all peripherals but the asynchronous counter and oscillator, which runs off an external 32kHz crystal.

The PIC18 has two power-saving modes – One runs the peripherals but not the core, the other powers down both (but allows the asynchronous timer to run). The 8051 device has similar power-saving modes. The PIC16 is even simpler – it only has one power-saving mode. This allows the asynchronous timer to run with everything else powered off, or the ADC (allowing a more accurate conversion due to lower system noise).

### 4.2 Clock Scaling

The PIC18, the ATMega128L and the MSP430 all implement software-controlled clock scaling.

The MSP430 uses a low-frequency analog oscillator to generate its base clock (at 32kHz) and then multiplies this using a digital frequency-locked loop (FLL). The multiplier factor can be set in software, allowing the clock to be set to any

value from 32kHz up to 6MHz (at 3V). The FLL can also be disabled, decreasing the accuracy of the generated clock (since it is now functioning as an open-loop multiplier) but saving power.

The 8051 uses a similar approach – using a phase locked loop (PLL) to generate a 12MHz clock and then dividing this down to give the system clock, from 6.3MHz (at 3V) down to 98kHz.

The ATMega128L implements a seven bit digital clock divider on the incoming clock signal. This affects the peripherals as well as the core. The core clock frequency must be at least four times the asynchronous clock frequency – giving a minimum core frequency of 128kHz.

The PIC18 can use an external crystal for precise timing, or an internal 8MHz RC oscillator. This can be divided down to any of eight set frequencies between 32kHz and the 4MHz. It can also be multiplied by four using a PLL to generate frequencies of up to 32MHz (although at 3V the maximum is 20MHz). The core can also use the same clock source as the asynchronous timer, running at 32kHz. Switching clocks results in a delay of two cycles of the old clock plus three cycles of the new clock. The PIC18 has a “two-speed startup” option, where the clock is provided by the internal RC oscillator immediately when the device wakes up, and then transitions to the selected clock when it has stabilised.

The PIC16 does not support any clock scaling. The clock frequency must remain constant, between DC and 10MHz (at 3V).

### 4.3 Memory Architecture

The MSP430 has a single (Von Neumann) address space, with data RAM and program ROM all accessed by a single 16-bit pointer. It supports a variety of addressing modes and has dedicated stack instructions, and a stack pointer register.

The ATMega128L uses a Harvard architecture. It has a dedicated stack pointer and three dedicated (sixteen bit) indirect memory access registers, accessed directly or using a constant offset.

The PIC18 also uses a Harvard architecture. It contains a dedicated, 32-level function call stack, and a “fast save” area – the interrupt logic saves the three most commonly used registers here. The data memory is divided into 256 byte banks. The current bank is accessed in one instruction, otherwise the bank must be switched. The first half of bank zero and the second half of bank fifteen (containing peripheral access registers) are mapped into a special “access bank” which is always accessible in a single instruction. Included in this are three dedicated indirect access registers. Accesses using a base and an offset must be done explicitly.

The PIC16 uses a simplified version of the PIC18 architecture. It has a maximum of 512 bytes of accessible RAM, divided into four 128 bytes banks, of which 386 bytes are general-purpose. 16 bytes of this are mirrored across every bank. Apart from one indirect access register, memory accesses must be within the current bank. The function call

stack is also fixed in hardware, and only contains eight levels. It has one accumulator which is targeted by most of arithmetic instructions.

The 8051 supports up to 2048 bytes of Extended RAM, and 256 bytes of normal RAM. However, only the first 128 bytes of normal RAM is accessible directly, and this overlaps the register space. The register space consists of four banks of eight registers. The next 16 bytes consists of bit-addressable memory. This leaves 80 bytes of general-purpose RAM in this bank. The remaining 128 bytes consists of SFRs, and is only accessible indirectly, as is the ERAM. The 8051 contains stack pointer registers and stack instructions.

Both the 8051 and the PIC16 were not designed with C code in mind, and C code compiled for these architectures tends to produce large program images, although previous work [7] suggests that the compiled data stack of the PIC C compiler deals well with this for simple applications.

#### 4.4 Instruction Set

Both the ATmega128L and MSP430 have rich instruction sets – a wide range of arithmetic instructions, dedicated multiply hardware and many addressing modes. The PIC18 and 8051 are more limited. One important factor for the PIC16 is the omission of a dedicated “add with carry” instruction, which makes even simple arithmetic on values greater than eight bits inefficient.

#### 4.5 I/O

All of the microcontrollers feature a number of digital I/O ports. These are multiplexed with peripheral hardware, such as UARTs and external timers, so in a particular application, not all of them will be available.

The MSP430 features 40 digital pins, the ATmega128L 53, the PIC18 36, the PIC16 33, and the 8051 37.

All of the devices contain a hardware UART. The ATmega128L has hardware supporting I<sup>2</sup>C and SPI, four timers and a ten bit, eight channel ADC. The MSP family has an extensive set of I/O options ranging from low-power devices with little I/O to devices with multiple ADCs, serial I/O support and even DMA controllers, although the particular device considered here does not have any of these. The PIC18 has hardware which can either carry out I<sup>2</sup>C or SPI, four timers and a thirteen channel, ten bit ADC. The PIC16 is the same, but only three timers and eight ADC channels. The 8051 has three timers, SPI support in hardware, and two independent 24-bit ADCs.

#### 4.6 Compiler

The MSP430 and ATmega128L are supported by a free port of gcc. The PIC16 and PIC18 have their own commercial compilers available, although the limitations of the load/store architecture make these relatively inflexible. The 8051 is supported by a number of C compilers, including SDCC, a free compiler. The existing compilers for the PIC architectures use a compiled stack, all others use a dynamic stack.

Commercial compilers for many of these architectures are

available, however at the time of writing the TinyOS build system is focused on a gcc-based compiler.

## 5. RESULTS

The current consumption figures of the various states of each microcontroller are listed in Table 1, taken from the appropriate datasheet. For the purposes of comparison, all of the figures are given at operating frequencies of 8MHz and 1MHz, with no peripherals enabled in active mode, and at the same frequencies with only an asynchronous timer enabled for power-down mode. Where the chip is capable of dynamically switching to a 32kHz clock in active mode this is given. The ATmega128L must operate at at least 128kHz, but the value at 32kHz is still given here for comparison. Similarly, the 8051 can only operate from 98kHz to 6.3MHz but values are extrapolated outside this range. All values are taken at an operating voltage of 3V. This does not give the energy used since different operations take different times on the various architectures.

The wakeup time is the time from an interrupt being signalled to the beginning of the interrupt handler – this does not include software overhead from saving registers etc. (see Section 3). The wakeup time for the PIC18 assumes two-speed startup is enabled – the initial part of the interrupt handler will execute with an RC oscillator, not the crystal oscillator if enabled.

Average current consumptions for a TDMA application are given in Table 2. The “Rx” application listens for a synchronisation signal once per second, “Rx+Tx” does this and also transmits a reply once per second, in the correct timeslot. The values used were measured on the PIC16 and AVR and estimated for the other architectures. Assuming all devices are powered by a 3V source, this is directly proportional to energy consumption since it takes into account the time the processor is actually active in each device.

A packet-level interface is assumed, so the microcontroller spends very little time in the active state. The most common operations are waking from sleep, setting an output pin, restarting a timer and returning to sleep, and using the SPI bus to shift out a packet to the radio chip. Of these operations, the only time the microcontroller is active is while restarting the timer. The power consumption is dominated by the wakeup time. These values are calculated with an operating frequency of 8MHz. The “Rx + Tx” application contains twice as many “wake + sleep” operations per second as the “Rx” application.

Table 3 shows the “Rx+Tx” application with the introduction of encryption, using values measured by P. Ganesan et al. [5]. The process of encryption consists of repeating simple mathematical operations and gives a good indication of the relative power consumption of the active modes of the various platforms. It is assumed that the 16-byte payload of the transmitted and received data must be encrypted and decrypted respectively. Two different algorithms are considered, RC5 and IDEA – the primary difference is that bitwise shifts are used in RC5 while multiplication is used in IDEA.

## 6. CONCLUSIONS

	AVR	PIC16	PIC18	MSP	8051
Word Size	8 bit	8 bit	8 bit	16 bit	8 bit
Max F at 3V	8Mhz	10MHz	20MHz	6MHz	6.3MHz
Power Down	8 $\mu$ A	20 $\mu$ A	2.6 $\mu$ A	1.8 $\mu$ A	21 $\mu$ A
Idle (1MHz)	0.5mA	220 $\mu$ A	120 $\mu$ A	55 $\mu$ A	n/a
Idle (8MHz)	4mA	1.5mA	843 $\mu$ A	440 $\mu$ A	n/a
Active (32k)	88 $\mu$ A	n/a	35 $\mu$ A	19.2 $\mu$ A	2.78mA
Active (1M)	2mA	220 $\mu$ A	480 $\mu$ A	240 $\mu$ A	4.05mA
Active (8M)	8mA	1.5mA	2.4mA	1.9mA	13.3mA
Wakeup	2ms	102 $\mu$ s	10 $\mu$ S	6 $\mu$ s	20 $\mu$ s

**Table 1: Current consumption information**

	AVR	PIC16	PIC18	MSP	8051
Rx	76.5 $\mu$ A	22.0 $\mu$ A	3.81 $\mu$ A	2.40 $\mu$ A	30.9 $\mu$ A
Rx + Tx	138.0 $\mu$ A	23.3 $\mu$ A	4.66 $\mu$ A	2.83 $\mu$ A	34.9 $\mu$ A

**Table 2: TDMA current consumptions**

While the microcontroller is a central part of any wireless sensor node design, little consideration has gone into choice of device in the past. There are a wide range of microcontrollers currently on the market – all of them offer broadly similar features. Node lifetime is determined by battery life, so power conservation is the prime concern.

The type of memory and memory access instructions determine how efficiently code will execute. An overhead of a few instructions per memory access can significantly influence the executing duty cycle, and power dissipation. More complex instruction sets can also speed up processing tasks, allowing the system spend more time sleeping. A useful feature is the ability to self-program. In order to generate efficient code, a well-optimised version of gcc or a similarly flexible, optimising compiler for the target architecture is a major advantage. However, due to the design of gcc it does not deal well with accumulator-based architectures such as the PIC.

Clock scaling can also be used to save power. Depending on the application, it may be more useful to run continuously at 32kHz and switch bits in the output without sleeping at all than run at 8MHz, waking up and sleeping. Clock flexibility tends to reduce power consumption.

In determining the power used by a sensing application (particularly using TinyOS) it is important to examine the most common operations. The “Timer-based state machine” approach is widely used to control external devices. While the system is only executing “useful” code for a single instruction, depending on the hardware it may require several milliseconds to power on and carry out the extra code required to handle an interrupt correctly.

In a typical application it can be seen that wakeup time

	AVR	PIC16	PIC18	MSP	8051
RC5	151 $\mu$ A	26.4 $\mu$ A	8.99 $\mu$ A	4.55 $\mu$ A	62.8 $\mu$ A
IDEA	148 $\mu$ A	27.8 $\mu$ A	8.26 $\mu$ A	7.56 $\mu$ A	75.1 $\mu$ A

**Table 3: Encryption current consumptions**

can dominate power consumption. When the number of wakeups was doubled the power consumption of the AVR, with the slowest wakeup time, almost doubles while for the MSP, with the fastest wakeup, it only increases by 18%.

The MSP family of microcontrollers from Texas Instruments have recently seen wider use for sensor networking applications and it is expected that the MSP will out-perform the older architectures in use until now.

## 7. REFERENCES

- [1] Analog Devices. *ADuC845/ADuC847/ADuC848 Datasheet, Rev. B*, 2005.
- [2] Atmel Corporation. *ATMega128 Datasheet, Rev. 2467I-09/03*, 2003.
- [3] A. J. M. et al. The Lutonium: A sub-nanojoule asynchronous 8051 microcontroller. In *proc. 9th IEEE Symposium on Asynchronous Circuits and Systems*, May 2003.
- [4] Eveready Battery Co. *Energizer No. 2450 Engineering Datasheet*.
- [5] P. Ganesan et al. Analyzing and modeling encryption overhead for sensor network nodes. In *WSNA '03: Proc. of the 2nd ACM int. conference on Wireless sensor networks and applications*, pages 151–159. ACM Press, 2003.
- [6] J. Hill. A software architecture supporting networked sensors. Master’s thesis, University of California, Berkeley, 2000.
- [7] C. Lynch and F. O. Reilly. Pic-based TinyOS implementation. In *proc. 2nd European Workshop on Wireless Sensor Networks, EWSN 2005, Istanbul*, pages 378–385, Feb. 2005.
- [8] A. Mainwaring et al. Wireless sensor networks for habitat monitoring. In *Proc. ACM Int. Workshop on Wireless Sensor Networks and Applications*, pages 88–97, Sept. 2002.
- [9] Microchip Technology Inc. *PIC16F877 Datasheet, Revision C*, 2000.
- [10] Microchip Technology Inc. *PIC18F2525/2620/4525/4620 Datasheet, Revision B*, 2004.
- [11] Nordic VLSI ASA. *Nordic nRF2401 Datasheet, Revision 1.1*, June 2004.
- [12] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design, The Hardware / Software Interface*. Morgan Kaufman, 2nd edition, 1998.
- [13] Texas Instruments. *MSP430C13x1 Datasheet, revised Sept. 2004*, 2004.