

SLIP group A report

Alan Anderson

Our team's assigned project was to design and implement a wireless sensor network for environmental monitoring and control of the Informatics Forum. Since I have worked with a variety of embedded platforms in the past, have a background in electronics and am proficient in C programming, I was assigned the task of building the wireless sensor network and coordinating messages going to and coming from the rest of the system.

Design

Wireless Sensor Network Design Considerations

A multi-hop design for our network is necessary, since it was likely that in a large building (such as the informatics forum) not all devices would be in range of all others. By using a multi-hop protocol, we can be sure that if a node can connect directly to one node of the network, it can communicate with all nodes on that network.

Although each node needs to be able to forward messages, there are no cases in our system where two sensor nodes communicate directly with one another – all communications either originate at or are directed towards the base station. All messages on the network are one-to-one, as we had no need of multicast messages.

The required data rate per node for the system is extremely low. In normal operation, each node is likely to be sending updates no more frequently than every five minutes, as most environmental sensor data changes slowly over time.. Message latency is also non-critical, meaning that it will be of little consequence if it takes several seconds for a sensor reading to be returned to the base station.

An additional constraint is that each node needed to be assigned a unique hardware address on the network, which cannot be dynamically allocated (similar to a computer network card's MAC address).

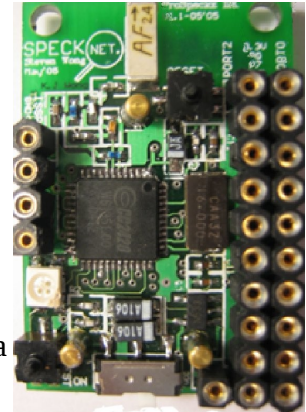
Hardware choices

All the hardware provided for the SLIP project had an IEEE 802.15.4 compliant radio transceiver, the standard used by the Zigbee wireless protocol. There were 3 types of wireless nodes made available:

Prospekz IIk

This is a small battery powered board made with off-the-shelf parts, with hardware documentation freely available. It consists of a Cypress Micro CY8C29666 programmable System on Chip (PSoC) and CC2420 RF transceiver. The PSoC has reconfigurable analogue and digital 'blocks', which can be used as ADCs, SPI master or slave ports, and UART / USARTs. The only way to interface the Prospekz hardware to a PC is via a serial port and some charge pump hardware.

The reconfigurable analogue blocks allow a wide range of sensors to be connected directly to the board. All firmware programming is done in C or ASM, and the gateway (block placement, wiring, etc) is modified through a GUI. The microcontroller core runs at a maximum of 16MHz.



SunSPOT

The SunSPOT hardware is unknown, except that it also has a CC2420 radio transceiver. All programming is done in Java, and no modification of the gateway is possible. There is a low-level API for the radio chip which allows advanced control. Communication with a computer is over USB, where it appears as an emulated serial port. It is possible to connect external input/output devices to the SunSPOT in addition to the built-in light sensor.

Jennic

The hardware is completely unknown, and can only be accessed via high-level APIs. The development boards made available to us had light, temperature and humidity sensors. The devices can be programmed in ANSI C, and communication with a PC is possible via either a serial or USB port.



Since all the hardware was capable of transmitting valid 802.15.4. data frames, all the hardware could be made to work together, given appropriate software support.

The SunSPOT was chosen to act as the bridge between the wireless sensor network (WSN) and the computer system since it can be trivially accessed. It appears as a serial COM port, so reading and writing to and from this base station node from within a program becomes very easy. There was also some example code for this purpose left from a previous year's SLIP project.

The prospekz IIk hardware was chosen as the platform to act as the first implementation of our chosen wireless protocol as detailed hardware documentation was readily available, and there was extensive example code available.

The Jennic hardware is built from ASICs made by Jennic, and as a result the end user only has access to programming APIs, rather than hardware documentation, as the case is if a board is built with off-the-shelf parts. Jennic boards have a full Zigbee implementation library, and encourage developers to use this API, rather than to implement their own custom protocol as we needed to. This did not allow sufficient access to the physical radio layer to implement a custom MAC protocol. As a result, we were unable to use the Jennic hardware.

SunSPOTs can also act as wireless sensor nodes, and communicate with the Prospekz nodes.

All of the MAC protocols considered for the system used Carrier Sense Medium Access (CSMA), meaning the radio hardware must have some mechanism of reporting whether or not the radio channel the node is using is currently being used by another node. Such systems are common, with Ethernet being a common simple example of CSMA: the all devices on the Ethernet bus wait until it is clear before they begin transmitting. Although non-CSMA MAC protocols are widespread, they are very collision prone, and are unreliable.

The radio chip in both the SunSPOT and Prospekz IIk is the Texas Instruments CC2420 (formerly manufactured by Chipcon). It has in-built hardware support for carrier-sense, which is exposed at the low level to the system architect.

The hardware for the Jennic boards is custom and proprietary, and although the hardware is capable of carrier-sensing, it is not exposed in the high-level API. This is the primary reason these development boards were discounted from being used as nodes.

Choosing the MAC protocol

The requirements for the MAC protocol were

- simple (due to limited system memory on the sensor node devices)
- scalable (to accommodate possibly very large numbers of devices)
- low power (to give longest possible battery life)

The choice of MAC protocol was also constrained by the particular hardware we had available, and what access we had to very low-level routines on that hardware.

Candidates for the MAC protocol included Zigbee, B-MAC, SpecMAC-B, and SpecMAC-D.

Although Zigbee is a protocol expressly designed with scalable wireless sensor networks in mind, it was quickly discounted from our list of possibilities. It is a non-trivial protocol, and only one of the three hardware devices available to us had a pre-written Zigbee library. Had we chosen to use it, we would have to have written two separate implementations for the different hardware architectures. It is also quite a signalling-heavy protocol, which is disadvantageous for extending battery life.

SpecMAC-B and B-MAC are very similar, in that they are both variations on a similar theme:

If a wireless node wishes to transmit a packet, it waits until it senses that no other nodes are transmitting, sends a long set of preambles, and follows them by one transmission of the data. At all other times, the node will periodically turn its receiver on briefly to sense if another node is transmitting preambles. If a transmission is detected, the receiving node keeps listening until the data at the end of the preambles is received. In the rare case that two nodes start to transmit at the same instant and their packets collide, both nodes will stop transmitting and 'back off' for a small random amount of time (much as happens in Ethernet). The first transmitter to begin transmitting is allowed then to send its entire message.

SpecMAC-D differs notably from the previous two protocols, as it replaces the preambles with multiple identical iterations of the packet being sent. This means that when a receiver wakes up periodically to sense the carrier for a transmission, it need only stay on as long as it takes to catch one complete iteration of the packet, so then it can return to low-power sleep mode. Since each

packet is transmitted many times, the packets must have a sequence number (much like TCP) to allow other nodes to distinguish between two identical sensor readings and the same packet being retransmitted.

Since SpecMAC-D turns the radio on for less time than either B-MAC or SpeckMAC-B, it results in an extended battery life. The simplicity of the algorithm used is ideal for embedded devices, and the rapid retransmission also later turned out to be well suited to the Prospekz IIk's radio hardware. The number of times each packet is retransmitted can be adjusted to give either extended battery life or improved message throughput.

An additional benefit to using a SunSPOT as the base station was that since it must be permanently connected to the host PC via USB, it is continuously powered.

The final hardware and MAC protocol choices are as follows:

Base station:	SunSPOT connected to a host PC over a USB link
Sensor nodes:	SunSPOT and Prospekz IIk
MAC protocol:	SpecMAC-D

Implementation

As mentioned in the previous section, one of the key aims of the WSN is to reduce power consumption as far as possible to extend node battery life. There are several general methods used in the embedded systems world to achieve low power operation:

- power down unneeded hardware
- operate at the lowest possible clock frequency to do the necessary work
- if possible, power down the system entirely until there is work to be done
- exploit hardware acceleration when possible

Gateway

To interface with other chips (eg. the CC2420 radio) and sensors, the PSoC needs to have its inputs, outputs and programmable blocks configured appropriately. This is done via a GUI tool in the PSoC Designer tool, where the user drags and drops digital and analogue blocks into free spaces, then routes the connections between them.

The Prospekz design used the following 15 digital blocks:

SPI Master and Slave(uses 2 blocks)

The Serial Peripheral Interface to the radio chip directly required two digital blocks, one to act as an SPI master (which generates the data clock) and the other as a slave (which is driven by its master's clock). The master mode is used the majority of the time, as it sends commands to the chip, and reads registers and status information. The only time the SPI slave digital block is used is when the controller requests to read the CC2420's RAM. In this case, the radio chip switches to be the bus master, and clocks out its memory contents to the PSoC which acts as a bus slave.

UART Tx and Rx (uses 2 blocks)

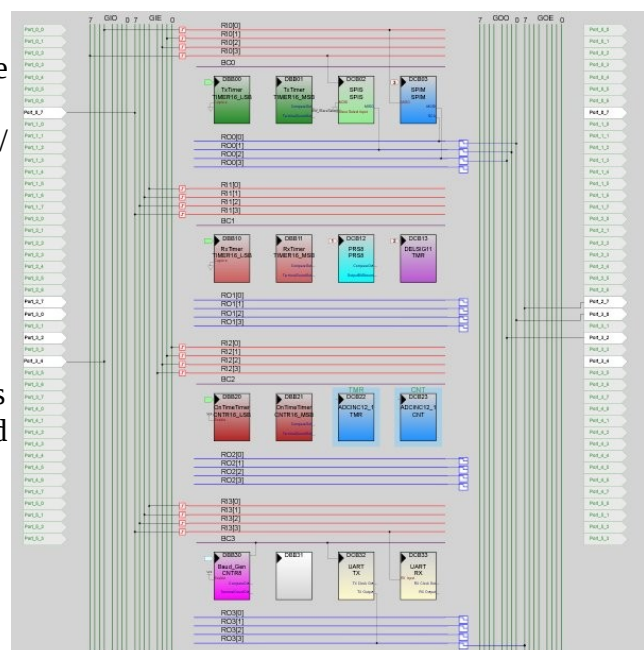
For communication with a computer, a two way UART connection over a serial port is possible. This was invaluable for debugging purposes, and could be useful in diagnosing problems in the field if a sensor node is thought to be defective.

Baud Generator (uses 1 block)

The UART blocks need a clock signal to stabilise the rate at which bits are clocked in and out. I chose a value for the generator to give a 9600 bit/s rate, which is a standard speed supported by all serial ports.

3 16-bit timers (uses 6 blocks)

There are 3 16-bit timers, each taking up two digital blocks. One is used by the software system to provide an interrupt at regular intervals (known as the 'system tick'). It can then be turned into a very versatile timer to schedule when events (such as sensor readings) are to happen. The second 16-bit timer is used by the SPI modules to time the clocking in and out of data. The third is required for a sigma-delta ADC in the analogue block.

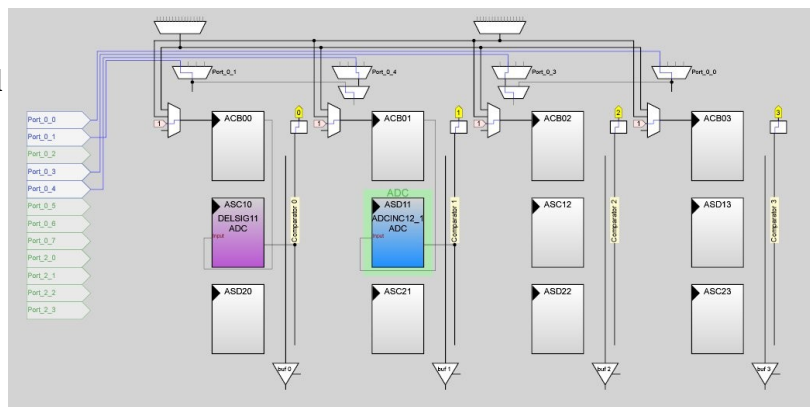


2 ADC digital units (uses 4 blocks)

The design uses two different kinds of ADC in the analogue section, and each ADC requires a clock module and a register to be read from.

The only analogue blocks I used were two ADCs. The first (purple in the diagram) was connected to an input multiplexer, so that I could connect it to one of 4 input pins under software control, meaning 4 sensors could be connected to the board, yet only need one ADC block. It is the ADC used for all the sensors connected to the board. Before the analogue voltage is passed into the ADC itself it passes through a Programmable Gain Amplifier (PGA) to

The second ADC (blue in the diagram) is connected statically to a single input pin which not connected to any other part of the Prospekz Iik circuit (a state known as 'floating'). This is generally considered bad practise, as the data being read from this block will have no meaning, since it is unconnected. However, having such a facility proves to be invaluable in one particular situation – when a random number generator



is needed. Reading an ADC whose input is left floating gives a limitless supply of high-quality random numbers, and requires no computation whatsoever.

SpecMAC-D requires a random number generator to negotiate backoffs: if two nodes start transmitting at precisely the same instant and their packets collide, they must delay by a random amount of time, and then attempt retransmission. Thus, one of the nodes will start transmitting first, and the other must wait until it has ceased.

Radio Driver and MAC protocol

Once SpeckMAC-D had been chosen, I began writing a driver for the Prospekz. My first test program found that the provided header files (“prospekz.h”) did not include support for any radio modes other than simple broadcasting and busy-wait receiving, meaning that the packet would be transmitted irrespective of how busy the channel was. Also, the receiving node would have a blocking call to a receive function, which would prevent any other work being done until a packet arrived. Below is some pseudocode to illustrate the point:

Transmitter	Receiver
<pre>int main(){ start_transmitter(); send_packet(NODE_B,"some message"); return(0); }</pre>	<pre>int main(){ start_receiver(); pkt= receive_packet(); //blocking call return(0); }</pre>

This kind of arrangement is clearly not suited to SpeckMAC-D, as it transmits irrespective of whether or not the channel is free. Also, having a blocking call to a function means that the processor must be running continuously, consuming power.

After further inspection, I found the radio hardware (CC2420) driver to be unsuitable for SpecMAC, and went about implementing my own driver.

The improvements over the existing driver include:

- enabled hardware support for clear channel assessment (CCA)
- enabled support for asynchronous receive and transmit
- changed message receiving mode from polling to interrupt driven
- added support for setting 802.15.4 radio channel via software
- changed register writes to using meaningful defined constants
- separated MAC and PHY layer code into different files

Moving to the new driver laid the foundation for being able to implement SpecMAC-D.

While rewriting the CC2420 driver I found that the radio hardware supports multiple transmissions of the same packet – it can store a packet in RAM once, and then transmit it many times, rather than having to transfer the same data from the microcontroller. The pseudocode comparison can be seen below:

Old Driver	New Driver
Packet pkt;	Packet pkt;
for(int i=0; i<100; i++){	send_packet(pkt); //writes out the packet once
send_packet(pkt);	retransmit(99); //requests 99 retransmissions
//writes out the packet 100 times	sleep(); //sleeps until done
}	...
...	

The obvious advantage of this mode is that while a node is retransmitting its packet many times, the microcontroller can be in sleep mode, even though its radio is actively transmitting. The radio then issued a hardware interrupt to the controller to wake up from sleep mode and to signal the transmission had finished.

Another useful feature of the CC2420 chip is that it has an on-board coulomb-counting battery meter. This allows the program running on the host microcontroller to find out how much power the battery has given out since it was last charged, allowing for a very accurate battery charge level indication. Although I included code for this feature in early versions of the driver, after inspection of the Prospekz hardware I found that the appropriate connections had not been made on the circuit board, leaving this feature redundant.

The CC2420 also allows for link quality indication (LQI), which allows a node to determine how strong a signal is from another node, and to assess how much RF noise there is on the channel. This feature is used extensively in Zigbee radio communications, and in certain situations can reduce power usage notably, or extend range. By using a series of simple algorithms, the nodes can agree between themselves the lowest transmit power necessary to reliably transmit data, and can even change frequencies entirely if there is excessive crosstalk on the channel currently being used.

It calculates the link quality by measuring signal strength, and comparing it to the number of correctly received RF 'chips' (where many 'chips' make up a bit).

We decided not to use the LQI simply because of the amount of additional signalling it would necessitate – instead we simply set each node to transmit with maximum power. Since most nodes will be communicating with each other less than every five minutes, LQI estimates would have changed by the next set of sensor readings were due to be transmitted.

Using maximum transmit power takes only twice as much current as using the minimum power, so the best case saving which could have been achieved by using complex signalling and LQI measurements is a factor of 2 during transmit power. Since such a time amount of the node's time is spent in this mode, the power savings would have been negligible, if at all once the additional signalling had been implemented.

As mentioned in the previous section, each node needs a unique hardware address for addressing on the network at the MAC layer. During development it was trivial to modify a constant in the source code for each device, however this would be impractical for more than a handful of devices. If there is no constant defined in the source code, the device reads the voltage levels at one of its 8-

bit input ports and uses that value as the least significant byte of the address. The pins can be pulled high or low by simply adding jumper leads to the Vcc or Gnd connections on the board.

By the time I had a working radio chip driver, I was given a previous student's code for a SpecMAC-D implementation, and was strongly encouraged to move to the new code base for the project, which I did. The new code was similar in design to my own, and completely interrupt driven. Although this is ideal from a power usage point of view, it means the program flow cannot be followed in any meaningful way.

After modifying the code to suit my purposes (changing the addressing mode, number of retransmissions, etc), I found that although the code was being compiled, no interrupts were being run. After huge amounts of wasted effort, it turned out to be down to a bug in the linker – interrupt handlers were simply not being linked, leaving a blank interrupt table. The code could be linked correctly by modifying the corresponding line in the ASM file.

By this stage, there was a working SpecMAC-D MAC layer for passing messages from one device to any adjacent device. The source code for the hardware driver and other noteworthy code is linked to off the same page as this report.

High level network protocols

Once the hardware and MAC layer had been successfully implemented, the messages being sent over the network needed to be defined, and routing information determined.

The MAC layer encapsulates a network layer message, which holds a collection of sensor readings from a single node. Each sensor reading contains a type descriptor (temperature, light, etc), a 32-bit node-unique timestamp followed by the sensor reading's value converted into appropriate units. The timestamp is to ensure that if the message is received twice (due to the mechanisms in SpecMAC-D) it will only be counted once.

The addressing on the network layer was designed to include routing details, but time pressures meant that this feature was neglected. As a result, the hardware address of each node was used as the network address. Routing was done via a simple flooding protocol – each message forwarded once by each node unless it was the recipient.

Sensor Hardware and conversion algorithms

The Prospekz controller is a PSoC with 16 analogue blocks, which can be configured as filters, comparators, ADCs, capacitive sensors and many other basic blocks. For my purposes, a 11-bit voltage ADC was sufficient. The hardware for the light and temperature sensing was trivial, needing only the most basic of potential divider circuits. This results in an analogue voltage output which can be directly read by the ADC.

Converting the 11-bit signed binary value into a meaningful unit of temperature or light intensity becomes more difficult, and huge errors are easily introduced.

There are three main sources of errors:

- sensor errors
- bias resistor errors
- quantisation errors

The temperature sensor I chose to use for the Prospekz modules has a process variation of 10%, meaning that two 'identical' sensors at the same temperature may differ in their readings by up to 10%.

The bias resistors used were accurate to within 10% of their stated value.

The quantisation error is the amount of uncertainty introduced into the reading by the ADC having to convert an analogue signal into one of several discrete values. In this case, the maximum quantisation error can be calculated to be $100/2^{11}$ %, which is about 0.05%, a trivial error when compared to the others introduced.

Adding these three errors give approximately a $\pm 20\%$ uncertainty about the actual value of the reading. The only way to overcome such huge errors is by manual calibration of each sensor, a tedious task.

The temperature sensor I chose has an approximately linear response to temperature, meaning that taking 2 ADC readings at known temperatures gave me a simple $y = mx + c$ relationship.

The light dependent resistor (LDR) on the other hand has a highly non-linear response, and as I had no equipment to measure light levels accurately, I was unable to calibrate the LDR. Light levels were expressed on a unit-less scale from zero (total darkness) to 100 (the maximum brightness the LDR could detect).

Analysis

By the end of the project we had produced a working wireless sensor network, with quite message passing, however there were a number of shortcomings:

Security

All sensor readings transmitted over the wireless network were sent in clear text, meaning that anyone with an 802.15.4 receiver could pick up all traffic. As environmental data is not sensitive data, this did not unduly concern us. A more potentially concerning attack is the injection of bogus packets onto the network. Suppose an attacker wished to misrepresent the temperature of an area of the building – he would only need to find the hardware address of the real node, then send packets appearing to come from that device. Misrepresenting the temperature of an area could cause the heating or cooling systems to be activated unnecessarily, potentially costing large sums of money.

This could be solved by having each packet cryptographically signed to verify the authenticity of the sender. This introduces an additional problem of key sharing, which would not be easily solved with the existing hardware and software solution, though it is by no means impossible.

Routing

As I have explained in previous sections, we used a very naive routing protocol. Given more time, a more advanced protocol could have been reached, which retained information about signal strengths from adjacent nodes, and made decisions about which path to take depending on least number of hops. Since fewer nodes would be used to transmit the data, it would mean extended battery life for all nodes. It would also have opened up the possibility of having more than one base station node on the network, which would further reduce the number of hops necessary to get a sensor reading back to a PC.

Sensor selection

As far as we developed the system, there is no way of changing which inputs are assigned to a particular sensor type, or how many sensors are physically attached to each node. Since this cannot be determined automatically, we would need to introduce a method for the user to tell a node which sensors were attached to which inputs. This could be done by introducing a new descriptor message, and the user could configure the sensor remotely.

Sensor errors

Using cheap analogue sensors result in huge errors as we found. Using more expensive digitally trimmed sensors is a viable alternative, although would require drivers to be written, and a re-design of some of the sensor request messages to reflect the new possibilities, since an arbitrary number of sensors can be connected to a single input pin (for examples, see Dallas-Maxim's 1-wire bus). However, power consumption would have to be considered before any radical hardware changes were made.

Scalability

Since we had only a small number of wireless nodes, we were unable to measure system performance as the number of sensors increased greatly.