Investigating a technique for programming wireless sensor networks

Philip Gordon Cass

Submitted in partial fulfilment of   
the requirements of Napier University   
for the Degree of  
Bachelor of Engineering with Honours in Computer Networks and Distributed Systems

School of Computing

May 2010

Authorship Declaration

I, Philip Gordon Cass, confirm that this dissertation and the work presented in it are my own achievement.

Where I have consulted the published work of others this is always clearly attributed;

Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;

I have acknowledged all main sources of help;

If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;

I have read and understand the penalties associated with Academic Misconduct.

I also confirm that I have obtained **informed consent** from all people I have involved in the work in this dissertation following the School's ethical guidelines

Signed:

Date:

Matriculation no:

Data Protection Declaration

Under the 1998 Data Protection Act, The University cannot disclose your grade to an unauthorised person. However, other students benefit from studying dissertations that have their grades attached.

Please sign your name below one of the options below to state your preference.

The University may make this dissertation, with indicative grade, available to others.

The University may make this dissertation available to others, but the grade may not be disclosed.

The University may not make this dissertation available to others.

Abstract

Wireless sensor networks are characterised, and some methods by which an application programmer could implement them are discussed. A relatively new programming technique called protothreads, which allows the programmer to separate the task into several distinct logical threads, although it is not a true multithreading implementation. After writing simple applications in protothreads, the attempt is made to implement the 802.15.4 wireless protocol. While not completed, protothreads are determined to have greatly simplified the implementation process.

Contents

[1 Introduction 8](#_Toc262724777)

[1.1 Aims, objectives and deliverables 10](#_Toc262724778)

[1.2 Dissertation structure 11](#_Toc262724779)

[2 Investigation of protothreads 12](#_Toc262724780)

[2.1 Effect of protothreads on source and complied code 12](#_Toc262724781)

[2.2 Protothreads test application 13](#_Toc262724782)

[3 Implementing a wireless networking protocol using protothreads 17](#_Toc262724783)

[3.1 The 802.15.4 standard 17](#_Toc262724784)

[3.2 Program design 23](#_Toc262724785)

[4 Results 25](#_Toc262724786)

[4.1 Beacon frame transmission 25](#_Toc262724787)

[4.2 Beacon frame reception and synchronisation 28](#_Toc262724788)

[4.3 Unimplemented features 29](#_Toc262724789)

[5 Evaluation of achievement 31](#_Toc262724790)

[6 Appendix 1 Code Listing 33](#_Toc262724791)

[6.1 Protothread example program 33](#_Toc262724792)

[6.2 802.15.4 network protocol – common files 42](#_Toc262724793)

[6.3 802.15.4 network protocol – coordinator 56](#_Toc262724794)

[6.4 802.15.4 network protocol – sensor node 60](#_Toc262724795)

[7 Appendix 2 Week 9 overview 66](#_Toc262724796)

[8 Appendix 3 Diary sheets 67](#_Toc262724797)

List of Figures

[Figure 1 Wireless Sensor Network in the home environment (4) 8](#_Toc262724618)

[Figure 2 Block diagram of wireless sensor node (5) 9](#_Toc262724619)

[Figure 3 Protothread example application – normal output 15](#_Toc262724620)

[Figure 4 Protothread example application - timer overflow 16](#_Toc262724621)

[Figure 5 OSI Model (12) 17](#_Toc262724622)

[Figure 6 General MAC frame format (13) 18](#_Toc262724623)

[Figure 7 Peer to peer network topology 19](#_Toc262724624)

[Figure 8 Star network topology 20](#_Toc262724625)

[Figure 9 Sequence diagram of communication in a beacon-enabled network 22](#_Toc262724626)

[Figure 10 Wireless sensor node program design 24](#_Toc262724627)

[Figure 11 Beacon frames with short interval 25](#_Toc262724628)

[Figure 12 Beacon frames with medium interval 26](#_Toc262724629)

[Figure 13 Beacon framess with medium interval and external oscillator 27](#_Toc262724630)

[Figure 14 Beacon frames with long interval and external oscillator) 28](#_Toc262724631)

Acknowledgements

I would like to thank Frank Greig, Alistair Armitage, and Aileen Maclean for their fantastic support during this project. In addition, I would like to thank Adam Dunkels for the creation of protothreads, upon which much of this work is based.

# Introduction

Wireless sensor networks (WSN) are set to change the way we live and work. At the present time, systems for reading domestic utility meters , for patient monitoring, for wildlife habitat monitoring and many other applications are either being developed or in use.

Figure 1 shows a typical WSN as it would apply in a domestic environment.

|  |
| --- |
| zigbee-home.PNG |
| Figure Wireless Sensor Network in the home environment |

The wireless node shown in Figure 1 indicates that there are two types of device; sensor nodes, which gather data from the environment and send it on, and coordinator nodes which collate the data and may forward it to a connected computer. Sensor nodes are typically not dependent on electrical sockets for power (i.e. contain a battery) while coordinator nodes may be plugged in, as usually they will be attached to a mains-powered computer anyway.

Typically a WS node consists as shown in Figure 2 of a microcontroller (a small computer on a single integrated chip) with a limited amount of permanent storage (flash ROM) and a very limited amount of RAM, connected to a radio transceiver module. The microcontroller should permit a variety of inputs and outputs, depending on application, including analogue to digital converters (ADC), digital to analogue converters (DAC), and digital I/O. (Sometimes called general-purpose I/O, or GPIO).

|  |
| --- |
|  |
| Figure Block diagram of wireless sensor node |

Since the hardware is resource constrained, it is important that the system code is compact, to conserve ROM, executes fast, due to the relatively slow CPU, and is frugal in terms or RAM consumption. As well as having the general property of fast code execution, the node must react in a timely manner to external events. This precludes the use of a conventional operating system as during a time-critical period, other programs might be scheduled. There are several alternatives that a system designer might employ to address this.

A Real-Time Operating System (RTOS) is one designed to eliminate the timing uncertainties of a general-purpose OS and either greatly increases the chances, or possibly guarantees, that high priority code will execute in a timely fashion. Optional OS features are either absent or implemented in a modular way so they can be removed if not required. The RTOS effectively becomes the program being executed, with the programmer’s application a subroutine of that program. This means that the burden of writing low-level network code is removed, but while specifically being written to minimise these factors, the downside of an RTOS is an unavoidable increase in code size and RAM usage (with a possible increase in node cost), and a loss of programmer control. The extra complexity also increases the difficulty of debugging, and the programmer must also spend time learning how to use the RTOS.

By contrast, a programmer might write the low-level routines himself, taking responsibility for juggling the different tasks a node must perform. Typically this is done by graphing the different possible states the node might be in, and constructing a program loop where at each iteration a different branch of the code is executed depending on the state. For complicated systems this can very quickly become unwieldy and error-prone, and may make the code difficult to modify or re-use.

In contrast to the traditional methods, a new approach introduced by Adam Dunkels called “protothreads” is beginning to be used. In this approach, compiler features are used to alter the structure of the source code so that it appears many different threads are being executed simultaneously, but without the memory overhead and concurrency issues of true multithreading.

## Aims, objectives and deliverables

The aims of this dissertation are to explore the potential for applying protothreads to WSNs. In particular, the following aspects will be considered

* Protothreaded architectures and how they might be applied to WSNs
* Development of an application using 8051 platform to apply protothreads for input and output

In order to achieve the aims, the following programme of work will

* Evaluate protothread software development techniques and program architecture
* Use protothreads to simplify the design and implementation of a proof of principle network using a beacon-enabled wireless protocol

## Dissertation structure

The structure is as follows...

* Protothread investigation
* Implementing 802.15.4 wireless networking using protothreads
* Results
* Evaluation of achievement

# Investigation of protothreads

Developed by Adam Dunkels of the Swedish Institute of Computer Science, protothreads allow the source code of a program to be separated out into separate “threads” which, conceptually, are executed in parallel. A variable associated with each thread tracks what line of the code in each thread is to be next executed, and when the thread is invoked, program execution resumes there instead of at the start of the function. This is implemented by C pre-processor macros to ensure the programmer’s source code remains as clear as possible. Unlike a general purpose OS’s threading model (which is known as pre-emptive multitasking) where a thread can be interrupted at any time, protothreads utilise the older and less common cooperative multitasking method – the points at which a thread may yield control to another are explicitly stated. In a general purpose OS, pre-emptive multitasking is considered an advantage as it ensures no single thread can monopolise the system; conversely, in an embedded system, the ability to know exactly when control may pass to another routine is a clear benefit, and eliminates many of the issues inherent in multi-threaded systems which must access a shared resource such as locking, race conditions and synchronisation.

## Effect of protothreads on source and complied code

Dunkels et al measured the effect of writing typical event-driven microcontroller code using protothreads. They found, in general, an increase in compiled code size of between 13 and 18 percent due to the expansion of the precompiler macros into actual C instructions, although one program could not be optimised in as efficient a manner as before, and grew by 72%. By contrast, a different program lent itself more to optimisation using protothreads, and compiled code actually *shrank* by 23%.

The number of machine instructions required to implement protothreads is shown to be approximately double that of a traditional event-driven method. However, this is still on the order of tens of instructions, so is negligible compared to the time required for activities like serial port communication.

The most important consideration, from the point of view of Dunkels et al, is that the source code of programs shrank, significantly, and this reduction was coupled by an increase in readability and clarity in the source code. In the slides associated with their presentation they indicate that the act of rewriting two of the programs to use protothreads exposed errors in the original implementations that were not obvious at the time.

Protothreads also have some technical limitations. Most drastically, variables local to a protothreads function do not save their state between calls. In cases where only one instance of a protothread will be running, this means the programmer must use static variables for storing the protothread’s data. In the case where a protothread may need to be called multiple times (where a server might have a thread for dealing with each client request, for example), this workaround cannot be used, as the static variable would be shared between each protothread. In that case, the main loop invoking each protothread would have to allocate storage for the protothread, and, in between calls, set a global variable to point the next protothread to the appropriate memory location. This limitation, plus the fact that the main loop would also need to manage the creation and expiration of the protothreads manually, makes protothreads unsuitable for this common design pattern.

A further limitation is that, due to the protothread macros use of the switch statement, the programmer cannot himself make use of the switch statement in a protothread. However, as the if statement can be used as an alternative (with a slight loss in source code readability), this can be worked around. The main issue is that Dunkels et al caution that the use of the switch statement can cause unpredictable behaviour that may not be flagged as an error by the compiler, meaning that forgetting this limitation might cause problems that aren’t immediately obvious.

## Protothreads test application

A small application was developed to show how the use of protothreads allowed many different events to be handled simultaneously.

### Development platform

For all practical examples developed, the c8051F121 development board from Silicon Labs was used. This features the 8051 microcontroller, which is a common embedded processor that, in large quantities and in some configurations, is available for less than a dollar. The development board, however, has a much richer set of features, including 8k of RAM, 128k of flash ROM, two analogue to digital converters (ADCs) and two digital to analogue converters (DAC). It also features a serial peripheral interface (SPI) bus, connecting the microcontroller to a wireless radio controller chip, the cc2420, which will be discussed later.

### Functions of the application

A test application with 8 protothreads was written. 4 LEDs were controlled by one protothread each, blinking the LEDs on and off at a different frequency. 4 other LEDs were controlled by one protothread, which turned a single LED on at a time, in sequence. Another thread simply incremented the number of seconds and minutes the board had been running, with no direct output. The penultimate protothread output the current value of one of the boards ADCs, connected to a dial-controlled variable resistor, as well as the time values from the previous protothread, at regular intervals via the board’s built-in serial interface. The final protothread partially re-implemented the functionality of the getkey function from the standard library – while invoking the original halts all other processing until a key is received from the serial interface, using a protothread wait allows the other tasks to continue. Upon receiving a character, the thread simply echoes it back to the computer via the serial port.

### Protothread timing problems

As almost all the protothreads involve waiting until a certain time, an interrupt-driven timer was used. The intention was that every time the timer routine fired (in the test application, 10000 times a second, although the actual accuracy required by the application was much lower) that a global “clock” variable be incremented. Each individual protothread could then wait for as long as it required by waiting until the clock variable had passed a certain value. However, the issue of integer overflow (where an integer of maximum value is incremented, resulting in an integer of minimum value being returned) made implementing this much more difficult than was anticipated.

Given a current clock value *cur* and a wait value of *wait*, it is simple to detect when *cur* + *wait* will result in an overflow, and adapt accordingly. However, an additional problem is when *cur* + *wait* will not overflow, but is close enough to the limit that there is a very good chance the clock variable itself will overflow before it can be compared. In future applications, a more sophisticated, centralised system was used, which consumed more memory, and required more processing time for the timer interrupt service routine, but which was not susceptible to these faults.

### Results

Figure 3 shows the output from the serial port on the development board. As expected, every second the value of the ADC is displayed. The time value does not increment evenly; as both the output and the clock are set to update simultaneously, the clock protothread and the ADC protothread race against each other, and the winner can vary, depending on which protothread executes first after the timer interrupt fires. This is an expected result is this situation, although to prevent confusion, a real application should take steps to prevent this by staggering timings. On the development board, the LEDs also lit up in sequence as expected.

|  |
| --- |
| protothread-test-realterm-normal.PNG |
| Figure Protothread example application – normal output |

As referred to previously, the method used for timing was inadequate, as can be seen in Figure 4. After working for several seconds, many protothreads will “freeze” because they are waiting for the timing variable to wrap around to a very large number again. It can be seen that the protothread driving ADC value output is still running normally, but the protothread incrementing the clock is not. The dysfunction is also mirrored in the LEDs, several of which stop flashing in sequence for several seconds. However, the basic aim – to show multiple different tasks being performed simultaneously – was met, and in particular, the reimplementation of the getchar function shows that protothreads can be useful for handling multiple inputs and outputs in parallel.

|  |
| --- |
| protothread-test-realterm-normal.PNG |
| Figure Protothread example application - timer overflow |

# Implementing a wireless networking protocol using protothreads

With the basic applicability of protothreads shown, the next step was to apply the techniques in the development of wireless sensor network applications. The development boards used for the initial testing also incorporate a cc2420 radio transceiver designed for implementing the 802.15.4 wireless networking standard. The ZigBee protocol used in examples in section 1 is based on 802.15.4, and the SimpliciTI protocol from Texas Instruments can also be used on 802.15.4 devices.

## The 802.15.4 standard

Published by the Institute of Electrical and Electronic Engineers (IEEE), the standard covers both the physical and data link layers in the traditional 7-layer networking model (Figure 5).

|  |
| --- |
| protothread-test-realterm-normal.PNG |
| Figure OSI Model |

The standard was revised in 2006; however as the cc2420 predates this, the implementation was based on the 2003 version.

### Physical layer

Three distinct frequency ranges are defined, of which the cc2420 supports one, the 2.4GHz band. The data is sent as a series of symbols, each of which encodes 4 bits of data. The 2.4GHz band supports a single data rate, 62,500 symbols per second. Thus the bitrate of the protocol is 250 kilobits per second. 16 distinct radio channels in the band are specified.

Aside from the above, most of the complexities of the physical implementation are handled automatically by the cc2420, and are not relevant to the implementation.

### Data Link layer

Each frame has an 8 bit prefix specifying the length of the frame (although only values up to 127 are valid) and the frame itself is then formatted as in Figure 6. The frame control field determines the type of frame (beacon, data, command, or acknowledgement), various flags for features such as acknowledgement requests, and also specifies source and destination addressing modes. These determine whether the relevant address fields should include a full 8 byte unique address, a 2 byte short address which has been assigned, or nothing at all. A source and destination PANid – from PAN, Personal Area Network - which uniquely identifies the network to distinguish it if other devices are using the same channel, can be specified in the same way.

|  |
| --- |
| protothread-test-realterm-normal.PNG |
| Figure General MAC frame format |

Command frames are used by nodes for coordination functions like associating with a particular PAN. The standard defines two types of devices, full function devices (FFDs) and reduced function devices (RFDs). RFDs are only required to implement a subset of all possible command frames. The node which organises the PAN, and to which association requests is sent, is known as the PAN coordinator.

### Network topologies and beacon-enabled networks

There are two possible network topologies defined; peer to peer (Figure 7), where any device can communicate with any other, and star (Figure 8), where all communication must be either from or to the coordinator. While this is a limitation, many WSNs only require this kind of communication, and so working around it may not be necessary. In addition, a star topology allows the use of a beacon-enabled network.

|  |
| --- |
|  |
| Figure Peer to peer network topology |

In a beacon-enabled network, the coordinator will transmit a beacon at regular intervals. This beacon frame will contain data about the interval between beacons (the beacon order), and what portion of the interval the coordinator’s radio will be active for (the superframe order). Nodes which wish to transmit data to the coordinator must wait until they receive a beacon, then transmit their frames during the active portion.

|  |
| --- |
|  |
| Figure Star network topology |

In a beacon-enabled network, the coordinator will transmit a beacon at regular intervals. This beacon frame will contain data about the interval between beacons, and what portion of the interval the coordinator’s radio will be active for. Nodes which wish to transmit data to the coordinator must wait until they receive a beacon, then transmit their frames during the active portion.

However, if the coordinator wishes to transmit data to the nodes, it must signal, in the beacon frame, that data is pending for a particular node. That node can then send a data request frame to the coordinator, at which point the coordinator can send the data frame. (Figure 9)

While this is a fairly involved procedure, the result is that, should a node not wish to send data, and should the coordinator not signal via the beacon that data is pending for the node, the node’s radio need only be active while receiving the beacon frame. As a full-size frame takes, at most, just over 4 milliseconds to be transmitted, the node’s radio can be off for a large proportion of the time, to save power.

The minimum beacon interval (and minimum active portion) is 15.36 milliseconds (or 960 symbol periods). The beacon frame format allows the size of these to be increased in powers of 2, up to 14. 2 to the power of 14 is 16,384, leading to a beacon interval of almost 4 minutes and 12 seconds! By tuning the active portion, the coordinator also has the ability to save power.

However, in many WSN designs the coordinator is not required to run on battery power. If this is the case, and the coordinator is also never required to send data frames to nodes, a beacon-enabled network is not necessary. Nodes could run with their radios permanently off and only turn them on as and when they need to send data to the coordinator. Nevertheless, if two-way communications are required, a beacon-enabled network has the potential to preserve the battery life of the nodes.

|  |
| --- |
|  |
| Figure Sequence diagram of communication in a beacon-enabled network |

## Program design

The protocol was broken down into 4 elements (Figure 10)

* Transmitting frames
* Receiving frames
* Reacting to command frames
* Application

### Transmitting frames

A coordinator must transmit a beacon, wait until the active period has expired (possibly sending frames in response to requests), turn off the radio, then wait until it’s almost time to send the next beacon, and turn the radio on again. This is a clear linear sequence of steps that lends itself well to being implemented as a protothread. For a leaf node, there is an analogous set of steps, except the protothread must wait until a beacon has been received, rather than sending one, and frames should automatically be sent at the correct time.

### Receiving frames

The initial act of reacting to the cc2420’s notice of incoming frames, and downloading the frame from the cc2420, is performed in an interrupt that is triggered as soon as a complete frame has been received. In this way, the cc2420’s buffer can be cleared as quickly as possible to reduce the chance of a buffer overflow. Depending on the type of frame received, the frame will be stored in separate structures for each frame type.

### Command frames and system management

This is a protothread which will consume and produce command frames, like PAN association requests etc.

### Application

Finally, the end-user’s application protothread(s), which will consume and produce data frames, and perform whatever additional input and output the programmer desires.

|  |
| --- |
|  |
| Figure Wireless sensor node program design |

# Results

## Beacon frame transmission

A packet sniffer was used to capture the radio transmissions from the development boards. Figure 10 shows the beacon frame being transmitted at regular intervals.

|  |
| --- |
| protothread-test-realterm-normal.PNG |
| Figure Beacon frames with short interval |

With the beacon set to the shortest possible time, the interval between beacons should be 15.36ms, or 15360 microseconds. The observed time is around 15340 microseconds.

In Figure 11, the beacon order is set to 5, which means the beacon interval will be 2^5 or 32 times longer. This is 491,520 microseconds, however the intervals being observed are around 487,800. Over all 11 received frames, the average interval (by taking the total time elapsed and dividing by 10) is 487,776, which makes the 11th frame almost 40ms ahead of schedule! This level of inaccuracy makes large beacon lengths impossible.

|  |
| --- |
| protothread-test-realterm-normal.PNG |
| Figure Beacon frames with medium interval |

To attempt to solve this, the 8051’s time sources were examined. The 8051 has two crystal oscillators available, an internal one which runs at 24.5MHz, and an external which runs at 8HMz, with the internal oscillator was being used to enable the faster CPU clock rate. The clock source was changed to the external oscillator and the test re-run.

Figure 12 shows the second test. Over 11 frames, the average interval was 491,516 which is only 4 milliseconds askew, out of almost 5 seconds of sampling. These tests were repeated with other nodes, with the same results. It is obvious the internal oscillator is not accurate enough for precision timing.

|  |
| --- |
| protothread-test-realterm-normal.PNG |
| Figure Beacon framess with medium interval and external oscillator |

For the final test, the beacon order was set to 12 i.e. 2^12 times (4096) longer than the default beacon. The results are shown in Figure 13. Note that the sequence numbers are not consecutive – the second frame was not successfully received by the packet sniffer. Regardless, the average interval was 62,913,736 microseconds. 15360 x 4096 is 62,914,560 which means the error is less than 1 millisecond in a little over 4 minutes.

|  |
| --- |
| protothread-test-realterm-normal.PNG |
| Figure Beacon frames with long interval and external oscillator) |

## Beacon frame reception and synchronisation

A node was configured to listen for beacon frames, decode the timing information therein, and turn its radio off and on at the correct time. To track this, the following behaviour was programmed

* Both coordinator and leaf node will light an LED to indicate when the radio is powered up
* Leaf node will also alternate the status if an LED with every beacon frame received
* Leaf node to alternate a third LED whenever the timer representing the time to next beacon has expired before the radio has been turned on (i.e. the node has likely missed the beacon frame)

The following behaviour was observed:

* Both coordinator and leaf node turned on their radio LED simultaneously. As the leaf node’s radio should only stay active if there are pending transmissions, it immediately went dark again, while the coordinator’s LED stayed lit for the duration of the active portion of the beacon interval.
* The leaf node’s second LED alternated between lit and unlit as the first LED flashed on.
* The leaf node’s third LED stayed unlit

This shows that the leaf node is capable of going into low power mode and waking up in time to receive the beacon frames.

## Unimplemented features

The full set of features required for a working wireless sensor network were **not** implemented. The next stage would be to implement sending of frames from the leaf nodes to the coordinator, during the beacon’s active period. After that, implementing frame reception on the coordinator would ensure 1-way communication of data from the leaf nodes to the coordinator.

Following this, some command frames could be implemented, starting with those necessary for the coordinator to send data frames to the leaf node as per the second example in Figure 9. Two-way communication would then be possible.

PAN association commands and the ability of the leaf node to scan different channels for the coordinator, could then be added. This would ensure that the WSN could automatically select the channel with the least radio interference to reduce errors. As the 2.4GHz band is shared with wi-fi and Bluetooth, among others, this would be an important feature, as the IEEE state that dynamic channel selection is one of the most important ways methods for coexisting with these standards.

# Evaluation of achievement

The initial analysis of protothreads and their confirmation of a viable technique for programming on microcontrollers was successful. The ability to separate out each task into separate functions instead of needing to interleave code in one master loop is an advantage that more than outweighs the modest increase in code size and execution time.

A functioning wireless sensor network was not implemented. However, the functions that were completed benefitted greatly from protothreads. Simultaneously managing tasks such as, for instance, keeping the radio turned on for the correct period of time, while processing incoming frames, while processing sensor input, while checking for a clear channel in order to transmit frames, would be much more difficult to manage without protothreads, and be a great deal harder to debug.

Even so, the time required to implement the basic structure of the program was massively underestimated. A combination of inexperience with embedded programming and C, and total ignorance in electronics, caused debugging problems, especially those involving external conditions such as buffer over- or underflows, to consume much more time than was planned.

With the basic structures in place, however, completion of the implementation is achievable, and, due to the modular nature of protothreaded code, once complete it should be easy to bolt different applications onto the wireless framework. This would give many of the benefits of an RTOS without the increase in complexity.

In summary, while the implementation was incomplete, the value of protothreads in this area has been shown.

References

1. **British Gas Media Relations.** British Gas plans two million smart meters in British homes by 2012. *Centrica plc website.* [Online] 29 March 2010. [Cited: 3 May 2010.] http://www.centrica.co.uk/index.asp?pageid=39&newsid=1970.

2. **Friedlos, Dave.** Electronic Underpants Help Caregivers Cope With Incontinence. *RFID Journal.* [Online] 30 April 2010. [Cited: 13 May 2010.] http://www.rfidjournal.com/article/view/7558/1.

3. **Mainwaring, Alan, et al.** *Wireless Sensor Networks for Habitat Monitoring.* 2003.

4. **ZigBee Alliance.** Telegesis and ZigBee Get Into Hot Water. *ZigBee Alliance.* [Online] 2009. [Cited: 25 May 2010.] http://www.zigbee.org/imwp/download.asp?ContentID=14518.

5. **Greig, Frank.** Block diagram of wireless sensor node. 2010.

6. *Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems.* **Dunkels, Adam, et al.** Boulder, Colorado : s.n., 2006. Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems.

7. **Dunkels, Adam, et al.** Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems (Talk slides). *Protothreads - Lightweight, Stackless Threads in C .* [Online] November 2006. [Cited: 13 May 2010.] http://www.sics.se/~adam/dunkels06protothreads.ppt.

8. **Silicon Laboritories Inc.** MCU Parametric Search. *Silicon Labs Website.* [Online] 15 May 2010. [Cited: 15 Mat 2010.] http://www.silabs.com/products/mcu/Pages/MCUParametricSearch.aspx.

9. **ZigBee Alliance.** Membership Requirements. *ZigBee Alliance.* [Online] 2010. [Cited: 18 May 2010.] http://www.zigbee.org/Join/MembershipRequirements.aspx.

10. **Texas Instruments.** SimpliciTI - RF Made Easy. *Texas Instruments.* [Online] 2008. [Cited: 19 May 2010.] http://www.ti.com/corp/docs/landing/simpliciTI/index.htm.

11. **International Organization for Standardization.** Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model. *International Organization for Standardization.* [Online] 15 06 1996. [Cited: 18 May 2010.] http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269\_ISO\_IEC\_7498-1\_1994(E).zip.

12. **Wikipedia.** OSI Model. *Wikipedia, The Free Encyclopedia.* [Online] 19 May 2010. [Cited: 20 May 2010.] http://en.wikipedia.org/w/index.php?title=OSI\_model&oldid=362989767.

13. **IEEE Computer Society.** Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs). *IEEE.org.* [Online] 1 October 2003. [Cited: 10 May 2010.] http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf.

# Appendix 1 Code Listing

## Protothread example program

### F121-proto-LED-UART-timer.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Author : Philip Cass

Last Modified : 20th May 2010 Created : 1st March,2008

File : F121-proto-LED-UART-timer.c.

Target Hardware : Silicon Labs - C8051F120

Tool chain : KEIL C51 V7.05

Version : 1.0.0

Description :

Protothreaded version :

The UART is configured to transmit data at 57600baud using the data

format 8-data,1 start, 1stop bits. The program executes a simple command processor that accepts single =

character commands over the UART serial port and responds by sending a variety of message formats to the PC =

comport.

A timer implements a system counter to enable timed events.

LED0 is controlled by a simple (but flawed) protothread which uses this counter to time itself

LED1 is an identical wait, but has the additional code to account for the counter wrap around

LED2 and 3 are controlled in the same manner

The other 4 LEDs are controlled by one protothread

The timer has been set at 10,000Hz in order to demonstrate the wrap-around problem

In reality this probably doesn't need to be above 1000Hz

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//-----------------------------------------------------------------------

// Includes

//-----------------------------------------------------------------------

#include <include-dirs.inc>

#include <c8051f120.h> // 8-bit SFR declarations

#include <C8051F120sfr16.H> // 16-bit SFR declarations

#include <stdio.h>

#include <pt.h>

//-----------------------------------------------------------------------

// Global CONSTANTS

//--------------------------------------------------------------------------

#define TRUE 0x01 // Value representing TRUE

#define LED0\_Delay 10000

#define LED1\_Delay 10000

#define LED2\_Delay 5000

#define LED3\_Delay 2500

#define LEDs\_lit\_Delay 7000

#define LEDs\_dark\_Delay 3000

#define seconds\_Delay 10000

#define UART\_Delay 10000

sbit StatusLED = P1^1; // Red LED (D5) on CC2420 Target Board

sbit LED0 = P2^0;

sbit LED1 = P2^1;

sbit LED2 = P2^2;

sbit LED3 = P2^3;

//-----------------------------------------------------------------------

// Function PROTOTYPES

//-----------------------------------------------------------------------

void SYSCLK\_Init (void);

void PORT\_Init (void);

void UART0\_Init (void);

void ADC0\_Init (void);

void Timer3\_Init (void);

//---------------------------------------------------------------------------

// Global VARIABLES

//----------------------------------------------------------------------------

unsigned int ADC\_Sample; // ADC0 current sample value

static struct pt pt1, pt2 ,pt3 ,pt4 ,pt5, pt6, pt7, pt8; // Protothread status variables

unsigned int clock;

unsigned int minutes,seconds;

unsigned int sectarget,secclock;

//---------------------------------------------------------------------

// Protothreads

//---------------------------------------------------------------------

static int proto\_LED0(struct pt \*pt)

{

// If we want to preserve the contents of the variable across thread blocking, it must be static

static unsigned int clock\_target; // This lets us compare against the current clock to perform timed waits

PT\_BEGIN(pt);

clock\_target = clock;

while(1) {

LED0 = ~LED0; // Do something (presumably more useful)

// This code is faulty. When clock\_target wraps around, the wait becomes unreliable

// See proto\_LED1 for the fix

clock\_target += LED0\_Delay; // A) Set the new target clocktick we want to wake up on or after

PT\_WAIT\_WHILE(pt, clock < clock\_target); // Wait while the clock is less than that value

}

PT\_END(pt);

}

static int proto\_LED1(struct pt \*pt)

{

// If we want to preserve the contents of the variable across thread blocking, it must be static

static unsigned int clock\_target; // This lets us compare against the current clock to perform timed waits

PT\_BEGIN(pt);

clock\_target = clock;

while(1) {

LED1 = ~LED1; // Do something (presumably more useful)

// Comment A) is a standard wait. The code marked by comments B) and C) are special cases

if ((clock\_target + LED1\_Delay) < clock\_target) { // B) If in the course of this wait we are going to wrap around...

PT\_WAIT\_UNTIL(pt, clock < clock\_target); // ...wait until after the wrap around before proceeding

}

clock\_target += LED1\_Delay; // A) Set the new target clocktick we want to wake up on or after

PT\_WAIT\_WHILE(pt, clock < clock\_target); // Wait while the clock is less than that value

sectarget=clock\_target;

secclock=clock;

}

PT\_END(pt);

}

static int proto\_LED2(struct pt \*pt)

{

// If we want to preserve the contents of the variable across thread blocking, it must be static

static unsigned int clock\_target; // This lets us compare against the current clock to perform timed waits

PT\_BEGIN(pt);

clock\_target = clock;

while(1) {

LED2 = ~LED2; // Do something (presumably more useful)

// Comment A) is the actual wait. The code marked by comment B) is a special case

if ((clock\_target + LED2\_Delay) < clock\_target) { // B) If in the course of this wait we are going to wrap around...

PT\_WAIT\_UNTIL(pt, clock < clock\_target); // ...wait until after the wrap around before proceeding

}

clock\_target += LED2\_Delay; // A) Set the new target clocktick we want to wake up on or after

PT\_WAIT\_WHILE(pt, clock < clock\_target); // Wait while the clock is less than that value

// This is also faulty - there is a special case where clock\_target is really high, and clock wraps around

// before we get to compare

}

PT\_END(pt);

}

static int proto\_LED3(struct pt \*pt)

{

// If we want to preserve the contents of the variable across thread blocking, it must be static

static unsigned int clock\_target; // This lets us compare against the current clock to perform timed waits

PT\_BEGIN(pt);

clock\_target = clock;

while(1) {

LED3 = ~LED3; // Do something (presumably more useful)

// Comment A) is the actual wait. The code marked by comment B) is a special case

if ((clock\_target + LED3\_Delay) < clock\_target) { // B) If in the course of this wait we are going to wrap around...

PT\_WAIT\_UNTIL(pt, clock < clock\_target); // ...wait until after the wrap around before proceeding

}

clock\_target += LED3\_Delay; // A) Set the new target clocktick we want to wake up on or after

PT\_WAIT\_WHILE(pt, clock < clock\_target); // Wait while the clock is less than that value

}

PT\_END(pt);

}

static int proto\_4LED(struct pt \*pt)

{

// If we want to preserve the contents of the variable across thread blocking, it must be static

static unsigned int clock\_target; // This lets us compare against the current clock to perform timed waits

static unsigned char current\_LED;

PT\_BEGIN(pt);

clock\_target = clock;

current\_LED = 0x10; // LED4's position in P2

while(1)

{

P2 = (P2 & 0x0F) | current\_LED; // Leave the bottom four LEDs as they are, and light up one of the top four

// A standard wait

if ((clock\_target + LEDs\_lit\_Delay) < clock\_target) {

PT\_WAIT\_UNTIL(pt, clock < clock\_target);

}

clock\_target += LEDs\_lit\_Delay;

PT\_WAIT\_WHILE(pt, clock < clock\_target);

P2 &= 0x0F; // Turn the top four LEDs off

// Another wait

if ((clock\_target + LEDs\_dark\_Delay) < clock\_target) {

PT\_WAIT\_UNTIL(pt, clock < clock\_target);

}

clock\_target += LEDs\_dark\_Delay;

PT\_WAIT\_WHILE(pt, clock < clock\_target);

// Shift the LED to light up along one

current\_LED = current\_LED << 1;

if (current\_LED == 0) current\_LED = 0x10;

}

PT\_END(pt);

}

static int proto\_keep\_time(struct pt \*pt)

{

// If we want to preserve the contents of the variable across thread blocking, it must be static

static unsigned int clock\_target; // This lets us compare against the current clock to perform timed waits

PT\_BEGIN(pt);

clock\_target = clock;

seconds = 0;

minutes = 0;

while(1) {

// A standard wait

if ((clock\_target + seconds\_Delay) < clock\_target) {

PT\_WAIT\_UNTIL(pt, clock < clock\_target);

}

clock\_target += seconds\_Delay;

PT\_WAIT\_WHILE(pt, clock < clock\_target);

seconds++;

if (seconds > 59) {

seconds -= 60;

minutes++;

}

}

PT\_END(pt);

}

static int proto\_UART\_INPUT(struct pt \*pt)

{

unsigned char c;

PT\_BEGIN(pt);

while(1)

{

// The following 3 lines are adapted from getkey.c from Keil

// TODO: Implement this and/or getchar as a child protothread

PT\_WAIT\_UNTIL(pt,RI0);

RI0 = 0;

c = SBUF0;

//We could do something interesting but let's just echo it back

putchar(c);

}

PT\_END(pt);

}

static int proto\_UART\_ADC(struct pt \*pt)

{

// If we want to preserve the contents of the variable across thread blocking, it must be static

static unsigned int clock\_target; // This lets us compare against the current clock to perform timed waits

PT\_BEGIN(pt);

clock\_target = clock;

while(1)

{

// A standard wait

if ((clock\_target + UART\_Delay) < clock\_target) {

PT\_WAIT\_UNTIL(pt, clock < clock\_target);

}

clock\_target += UART\_Delay;

PT\_WAIT\_WHILE(pt, clock < clock\_target);

SFRPAGE = ADC0\_PAGE; // Start another conversion

AD0INT=0;AD0BUSY=1; // Start conversion

while(!AD0INT); // Poll/Wait for end-of-conversion

ADC\_Sample =(ADC0H << 8) + ADC0L; // Read ADC0 8-bit registers and form a 12-bit result

SFRPAGE = UART0\_PAGE;

printf (" ADC Sample >> %u\n", ADC0); // send result to serial port

printf (" Board running for %u minute(s) and %u seconds\n", minutes,seconds ); // send result to serial port

}

PT\_END(pt);

}

//-------------------------------------------------------------------------

// MAIN Routine

//---------------------------------------------------------------------------

void main (void) {

SYSCLK\_Init(); // initialize oscillator

PORT\_Init (); // initialize crossbar and GPIO

UART0\_Init (); // initialize UART0

ADC0\_Init ();

Timer3\_Init();

SFRPAGE = CONFIG\_PAGE;

StatusLED = ~StatusLED; // Toggle Status LEDs on P3.7

P3 = 0x00; // Set P3\_6 =3D 0 to enable ADC0 input potentiometer.

PT\_INIT(&pt1);

PT\_INIT(&pt2);

PT\_INIT(&pt3);

PT\_INIT(&pt4);

PT\_INIT(&pt5);

PT\_INIT(&pt6);

PT\_INIT(&pt7);

PT\_INIT(&pt8);

EA = 1; // Enable Global interrupts

EIE2 |= 0x01; // Enable Timer3 interrupts

SFRPAGE = UART0\_PAGE;

puts("Initialise Complete\n ");

while (TRUE)

{

proto\_LED0(&pt1);

proto\_LED1(&pt2);

proto\_LED2(&pt3);

proto\_LED3(&pt4);

proto\_UART\_ADC(&pt5);

proto\_4LED(&pt6);

proto\_keep\_time(&pt7);

proto\_UART\_INPUT(&pt8);

} // end while

} // end main

//-----------------------------------------------------------------------------

// Timer3\_ISR

//-----------------------------------------------------------------------------

void Timer3\_ISR (void) interrupt 14

{

SFRPAGE = TMR3\_PAGE;

TMR3CN &= ~0x80; // Reset TF3 flag

clock++;

if (clock % 100 == 0) StatusLED = ~StatusLED;

}

### MCUInit\_V0.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Author : Frank Greig

Last Modified : 28th February,2008 Created : 28th February,2008

Modified : Philip Gordon Cass 6th March, 2008

File : MCU\_InitV0.c

Target Hardware:Silicon Labs - C8051F120

Tool chain : KEIL C51 V7.05

Version : 1.0.0

Description :

MCU Initialisation routines for C8051F121 MCU Interface Module

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//-----------------------------------------------------------------------------

// Includes

//-----------------------------------------------------------------------------

#include <include-dirs.inc>

#include <c8051f120.h> // SFR declarations

#include <C8051F120sfr16.H> // 16-bit SFR declarations

//-----------------------------------------------------------------------------

// Global CONSTANTS

//-----------------------------------------------------------------------------

#define SYSCLK 24500000 // On-chip Internal oscillator frequency in Hz

#define BAUDRATE 57600 // Baud rate of UART in bps

#define T3\_Overflow\_Rate 10000 // Timer3 Overflow frequency in Hz

// If SYSCLK / overflow rate is greater than 65535, the numbers will overflow

// and timer frequency will not be as expected. At 24.5MHz clock, this means

// you need a timer rate of at least 374Hz - PGC

//-----------------------------------------------------------------------------

// Global Variables

//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------

// Initialization Subroutines

//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------

// Timer3\_Init

//-----------------------------------------------------------------------------

// Configure Timer3 to auto-reload at interval specified by <counts> (no

// interrupt generated) using SYSCLK as its time base.

//

void Timer3\_Init (void)

{

char SFRPAGE\_SAVE = SFRPAGE; // Save Current SFR page

SFRPAGE = TMR3\_PAGE;

TMR3CN = 0x00; // Stop Timer3; Clear TF3;

TMR3CF = 0x08; // use SYSCLK as timebase

RCAP3 = -(SYSCLK/T3\_Overflow\_Rate); // Init reload values

TMR3 = RCAP3; // set to reload immediately

EIE2 &= ~0x01; // disable Timer3 interrupts

TR3 = 1; // start Timer3

SFRPAGE = SFRPAGE\_SAVE; // Restore SFR page

}

//-----------------------------------------------------------------------------

// ADC0\_Init

//-----------------------------------------------------------------------------

// Configure ADC0 to use by write to AD0BUSY flag as conversion source,

// ADC Interrupt is generate an interrupt on conversion complete, and to use left-justified

// output mode. Enables ADC end of conversion interrupt. Leaves ADC disabled.

//

void ADC0\_Init (void)

{

char SFRPAGE\_SAVE = SFRPAGE; // Save Current SFR page

SFRPAGE = ADC0\_PAGE;

ADC0CN = 0x00; // ADC0 disabled; normal tracking

// mode; ADC0 conversions are initiated

// by write to AD0BUSY flag; ADC0 data is

// right-justified

REF0CN = 0x07; // enable temp sensor, on-chip VREF,

// and VREF output buffer

AMX0SL = 0x01; // Select potentiometer input on AMUX Channel 1

ADC0CF |= 0x20; // PGA gain = 0.15 and SARclock=2MHz = (22.11/(2x2) - 1 = 4

AD0EN = 1; // ADC enable

SFRPAGE = SFRPAGE\_SAVE; // Restore SFR page

}

//-----------------------------------------------------------------------------

// SYSCLK\_Init

//-----------------------------------------------------------------------------

// This routine initializes the system clock to use the internal oscillator

// at 24.5 MHz

//

void SYSCLK\_Init (void)

{

int i; // software timer

char SFRPAGE\_SAVE = SFRPAGE; // Save Current SFR page

WDTCN = 0xde; // disable watchdog timer

WDTCN = 0xad;

SFRPAGE = CONFIG\_PAGE; // set SFR page

OSCXCN = 0x06; // Disable the external oscillator with 8MHz crystal.

OSCICN = 0x83; // Enable the internal oscillator.

for(i=0; i < 256; i++); // Wait for the oscillator to start up.

while(!(OSCICN & 0x40)); // Check to see if the Internal Oscillator Valid Flag is set.

CLKSEL = 0x00; // SYSCLK derived from the Internal Oscillator circuit.

SFRPAGE = SFRPAGE\_SAVE; // Restore SFR page

}

//-----------------------------------------------------------------------------

// PORT\_Init

//-----------------------------------------------------------------------------

//

// This routine configures the crossbar and GPIO ports.

//

void PORT\_Init (void)

{

char SFRPAGE\_SAVE = SFRPAGE; // Save Current SFR page

SFRPAGE = CONFIG\_PAGE; // set SFR page

XBR0 = 0x06; //0xE7;

XBR1 = 0x00; //0x75;

XBR2 = 0x44; //0x5D; // Enable crossbar and weak pull-up and enable UART1

P0MDOUT |= 0x35; // Set SPI and TX1 pin to push-pull

P1MDOUT |= 0xC2; // Set CC2420 control signals/status

P2MDOUT |= 0xFF; // Set P2 as Outputs for LEDs

P3MDOUT |= 0xF0; // Set outputs and switches

SFRPAGE = SFRPAGE\_SAVE; // Restore SFR page

}

//-----------------------------------------------------------------------------

// UART0\_Init

//-----------------------------------------------------------------------------

// Configure the UART1 using Timer1, for <Baudrate> and 8-N-1.

// Note this only work for BAUD rate greater than 19,200 because the timer

// divider is only 8-bits

void UART0\_Init (void)

{

char SFRPAGE\_SAVE = SFRPAGE; // Save Current SFR page

SFRPAGE = UART0\_PAGE;

SCON0 = 0x50; // SCON0: mode 1, 8-bit UART, enable RX

SSTA0 = 0x10; // Select Timer 1 as buad rate generator

SFRPAGE = TIMER01\_PAGE;

TMOD &= ~0xF0;

TMOD |= 0x20; // TMOD: Timer 1, mode 2, 8-bit reload

TH1 = -(SYSCLK/BAUDRATE/16);

CKCON |= 0x10; // T1M = 0; SCA1:0 = 10

TL1 = TH1; // Initialize Timer1

TR1 = 1; // Start Timer1

SFRPAGE = UART0\_PAGE;

TI0 = 1; // Indicate TX1 ready

SFRPAGE = SFRPAGE\_SAVE; // Restore SFR page

}

## 802.15.4 network protocol – common files

### MCU\_Init8MHz\_V0.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Author : Frank Greig

Modified by : Philip Cass

Last Modified : 20th May,2010 Created : 17th July,2008

File : MCU\_Init.c

Target Hardware : Silicon Labs - C8051F121 - IEEE802.15.4 Target Board

Tool chain : KEIL C51 V7.05

Version : 1.0.0

Description :

MCU Initialisation routines for C8051F121 MCU - CC2420 IEEE802.15.4 Target Board

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//-----------------------------------------------------------------------------

// Includes

//-----------------------------------------------------------------------------

#include "..\Headers\SYS\_DEFS.H" // SiLabs COF8051F121 MCU functions and Macros

//-----------------------------------------------------------------------------

// Global Variables

//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------

// Initialization Subroutines

//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------

// SPI0\_Init

//-----------------------------------------------------------------------------

// Return Value : None

// Parameters : None

//

// Configures SPI0 to use 4-wire Single-Master mode. The SPI timing is

// configured for Mode 0,0 (data centered on first edge of clock phase and

// SCK line low in idle state). The SPI clock is set to 4MHz approx.

// The NSSMD0 pin is set to 1.

//-----------------------------------------------------------------------------

void SPI0\_Init(void)

{

char SFRPAGE\_SAVE = SFRPAGE; // Save Current SFR page

SFRPAGE = SPI0\_PAGE;

SPI0CFG = 0x40; // Enable SPI master mode; clocked on rising edge; SCLK is idle low.

SPI0CN = 0x0D; // 4-wire single master mode; SPI enabled, NSSDM0=1

#if INTERNAL\_CLOCK

SPI0CKR = 1; // Frequency of SPIclk = SYSCLK/(2 \* (SPIOCKR +1)) 24.5/(2\*(1+1)) = 6.75MHz approx

#else

SPI0CKR = 0; // Frequency of SPIclk = SYSCLK/(2 \* (SPIOCKR +1)) 8/(2\*(0+1)) = 4MHz approx

#endif

SFRPAGE = SFRPAGE\_SAVE; // Restore SFR page

}

//-----------------------------------------------------------------------------

// ADC0\_Init

//-----------------------------------------------------------------------------

// Configure ADC0 to use by write to AD0BUSY flag as conversion source,

// ADC Interrupt is generate an interrupt on conversion complete, and to use left-justified

// output mode. Enables ADC end of conversion interrupt. Leaves ADC disabled.

//

void ADC0\_Init (void)

{

char SFRPAGE\_SAVE = SFRPAGE; // Save Current SFR page

SFRPAGE = ADC0\_PAGE;

ADC0CN = 0x00; // ADC0 disabled; normal tracking

// mode; ADC0 conversions are initiated

// by write to AD0BUSY flag; ADC0 data is

// right-justified

REF0CN = 0x07; // enable temp sensor, on-chip VREF,

// and VREF output buffer

AMX0SL = 0x01; // Select potentiometer input on AMUX Channel 1

ADC0CF |= 0x20; // PGA gain = 0.15 and SARclock=2MHz = (22.11/(2x2) - 1 = 4

AD0EN = 1; // ADC enable

SFRPAGE = SFRPAGE\_SAVE; // Restore SFR page

}

//-----------------------------------------------------------------------------

// SYSCLK\_Init

//-----------------------------------------------------------------------------

// This routine initializes the system clock to use the internal oscillator

// at 24.5 MHz multiplied by two using the PLL.

//

void SYSCLK\_Init (void)

{

int i; // software timer

char SFRPAGE\_SAVE = SFRPAGE; // Save Current SFR page

SFRPAGE = CONFIG\_PAGE; // set SFR page

WDTCN = 0xde; // disable watchdog timer

WDTCN = 0xad;

OSCICN = 0x83; // Configure internal oscillator for its highest frequency (24.5 MHz)

RSTSRC = 0x04; // Enable missing clock detector

#if INTERNAL\_CLOCK

for(i=0; i < 256; i++); // Wait for the oscillator to start up.

while(!(OSCICN & 0x40)); // Check to see if the Internal Oscillator Valid Flag is set.

CLKSEL = 0x00;

OSCXCN = 0x06; // Disable the external oscillator with 8MHz crystal.

#else

OSCXCN = 0x66; // Start external oscillator with 8MHz crystal.

for(i=0; i < 15000; i++); // Wait 1ms for the oscillator to start up.

while ((OSCXCN & 0x80) != 0x80); // Check to see if the Crystal Oscillator Valid Flag is set.

CLKSEL = 0x01; // SYSCLK derived from the External Oscillator circuit.

OSCICN = 0x00; // Disable the internal oscillator.

#endif

SFRPAGE = SFRPAGE\_SAVE; // Restore SFR page

}

//-----------------------------------------------------------------------------

// PORT\_Init

//-----------------------------------------------------------------------------

//

// This routine configures the crossbar and GPIO ports.

//

void PORT\_Init (void)

{

char SFRPAGE\_SAVE = SFRPAGE; // Save Current SFR page

SFRPAGE = CONFIG\_PAGE; // set SFR page

XBR0 = 0x16; // Enable CEX0,CEX1,UART0 and SPI

XBR1 = 0x04; // Enable INT0 only;

XBR2 = 0x44; // Enable Xbar and UART1

P0MDOUT |= 0x35; // Set NSS,MOSI,SCLK and TX0 pin to push-pull.TX1 disable set as input

P1MDOUT |= 0xC2; // Set VReg,Resetn,StatusLEd CC2420 control signals/status

P2MDOUT |= 0xFF; // Set P2 as Outputs for LEDs

P3MDOUT |= 0x70; // Set pot ground reference as output(P3^6), Switches as inputs

// Set USB suspend detect as input -P3^7

SFRPAGE = SFRPAGE\_SAVE; // Restore SFR page

}

//-----------------------------------------------------------------------------

// UART0\_Init

//-----------------------------------------------------------------------------

// Configure the UART1 using Timer1, for <Baudrate> and 8-N-1.

// Note this only work for BAUD rate greater than 19,200 because the timer

// divider is only 8-bits

void UART0\_Init (void)

{

char SFRPAGE\_SAVE = SFRPAGE; // Save Current SFR page

SFRPAGE = UART0\_PAGE;

SCON0 = 0x50; // SCON0: mode 1, 8-bit UART, enable RX

SSTA0 = 0x10; // Select Timer 1 as buad rate generator

SFRPAGE = TIMER01\_PAGE;

TMOD &= ~0xF0;

TMOD |= 0x20; // TMOD: Timer 1, mode 2, 8-bit reload

TH1 = -(SYSCLK/BAUDRATE/16);

CKCON |= 0x10; // T1M = 0; SCA1:0 = 10

TL1 = TH1; // Initialize Timer1

TR1 = 1; // Start Timer1

SFRPAGE = UART0\_PAGE;

TI0 = 1; // Indicate TX1 ready

SFRPAGE = SFRPAGE\_SAVE; // Restore SFR page

}

//-----------------------------------------------------------------------------

// Timer3\_Init

//-----------------------------------------------------------------------------

// Configure Timer3 to auto-reload at interval specified .

//

void Timer3\_Init (void)

{

char SFRPAGE\_SAVE = SFRPAGE; // Save Current SFR page

SFRPAGE = TMR3\_PAGE;

TMR3CN = 0x00; // Stop Timer3; Clear TF3;

// With neither clock is 1 second's worth of SYSCLK divisible by 12

// So use SYSCLK instead; this forces a minimum timer HZ of 690, which is fine

TMR3CF = 0x08; // Use SYSCLK as timebase

RCAP3 = -(SYSCLK/T3\_Overflow\_Rate); // Init reload values

EIE2 &= ~0x01; // Disable Timer3 interrupts

TR3 = 1; // Start Timer3

SFRPAGE = SFRPAGE\_SAVE; // Restore SFR page

}

### PHY\_CommsV2.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Author : Frank Greig

Modified : Philip Cass

Last Modified : 26th May, 2010 Created : 18th February, 2009

Filename : PHY\_CommsV1.c

Target Hardware : Silicon Labs - C8051F121TB

Tool chain : KEIL C51 6.03 / KEIL EVAL C51

Version : 1.0.0

Description :

Packet Initialisation Transmit and Receive routines for CC2420

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//-----------------------------------------------------------------------------

// Includes

//-----------------------------------------------------------------------------

#include "..\Headers\SYS\_DEFS.H" // System type definitions and all includes

xdata BYTE IEEE\_address[8];

void LoadPacket(Packet \*pkt){

BYTE bytecounter,intStatus;

SFRPAGE = SPI0\_PAGE;

intStatus = EX0; //save current status of int0

EX0 = 0; //regardless, turn int0 off

NSSMD0 = 0; // Enable Slave Select

if (SPIO(CC2420\_SNOP) & BM(CC2420\_TX\_UNDERFLOW)) { // Flush TX underflow

SPIO(CC2420\_SFLUSHTX);

}

SPIO(CC2420\_TXFIFO); // Enable Write to TX FIFO

SPIO(pkt->DataLength + PKT\_FOOTER\_LEN); // Packet Length is (Data + FCS (2B))

for (bytecounter=0; bytecounter<(pkt->DataLength); bytecounter++) // Write FIFO payload data bytes only

SPIO(pkt->PacketData[bytecounter]); // Write payload data to FIFO

NSSMD0 = 1; // FIFO access must terminate with - Disable CC2420

EX0 = intStatus;

}

/\* ---------------------------------------------------------------------------

Description : Initialise CC2420 Transceiver

--------------------------------------------------------------------------- \*/

void Initialise\_CC2420(){ // Initialise the CC2420

SFRPAGE = CONFIG\_PAGE;

VREGen = 1; // Set voltage regulator active

Delay (50);

RESETn = 0; // Reset CC2420 with low signal

Delay (1000);

RESETn = 1; // Bring CC2420 to normal operation

Delay (100);

SFRPAGE = SPI0\_PAGE;

NSSMD0 = 0; // Enable CC2420 Chip Select

WriteRegister(CC2420\_MDMCTRL0,0x0A72); // ADR\_DECODE = 1; AUTOCRC=1, 4:AUTOACK=1, 3:0:PL=2

WriteRegister(CC2420\_MDMCTRL1,0x0500); // Set correlator threshold =20

WriteRegister(CC2420\_IOCFG0,0x067F); // Set RxFIFO threshold to 127 (1 full packet) and invert FIFOP polarity so !>0 IRQ on INT1

SPIO(CC2420\_SXOSCON); // Turn on 16MHz oscillator

do{ // Wait for the XTAL oscillator to stablilise

} while (!( SPIO(CC2420\_SNOP) & (BM(CC2420\_XOSC16M\_STABLE)) ));

NSSMD0 = 1; // Disable CC2420 Chip Select

LongAddress(IEEE\_address,FALSE); // Read the 64-bit IEEE address from cc2420

}

/\* ---------------------------------------------------------------------------

Description :

Sets CC2420 for a given IEEE 802.15.4 channel.

Note that SRXON, STXON or STXONCCA must be run for the new channel selection to take full effect.

---------------------------------------------------------------------------- \*/

void SetRFChannel(BYTE channel) {

BYTE int3status;

WORD f;

// Derive frequency programming from the given channel number

f = (WORD) (channel - 11); // Subtract the base channel

f = f + (f << 2); // Multiply with 5, which is the channel spacing

f = f + 357 + 0x4000; // 357 is 2405-2048, 0x4000 is LOCK\_THR = 1

SFRPAGE = SPI0\_PAGE;

int3status = EIE2 & 0x01; //save current status of int3

EIE2 = EIE2 & 0xFE; //regardless, turn int3 off

NSSMD0 = 0; // Enable CC2420 Chip Select

WriteRegister(CC2420\_FSCTRL, f); // Write it to the CC2420

NSSMD0 = 1; // Disable CC2420 Chip Select

EIE2 = EIE2 | int3status;

} // SetRFChannel

/\* ---------------------------------------------------------------------------

Description : Sets the radio transmission strength

--------------------------------------------------------------------------- \*/

void SetTxStrength(BYTE strength)

{ //Set to Radio Power

BYTE int3status;

if (strength >31) strength= 0x1F; // Set to max power

SFRPAGE = SPI0\_PAGE;

int3status = EIE2 & 0x01; //save current status of int3

EIE2 = EIE2 & 0xFE; //regardless, turn int3 off

NSSMD0 = 0; // Enable CC2420 Chip Select

WriteRegister(CC2420\_TXCTRL, ((ReadRegister(CC2420\_TXCTRL) & 0xFFE0 ) | strength));

NSSMD0 = 1; // Disable CC2420 Chip Select

EIE2 = EIE2 | int3status;

}

### SPI\_CommsV1.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Author : Frank Greig

Modified : Philip Cass

Last Modified : 20th May, 2010 Created : 1st March,2008

Filename : SPI\_CommsV1.C

Target Hardware : Silicon Labs - C8051F120

Tool chain : KEIL C51 6.03 / KEIL EVAL C51

Version : 1.0.0

Description :

Physical Layer - CC2420 Wireless Communications Functions.

These routines implement the register access and control functions between

the MCU and CC2420 Wireless Transceiver.

Comments : Chip select and deselect are removed form the function primitives.

This allows strings of SPIO operations to be used without continualy

toggling the CS control signal - more efficient and probably safer.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//-----------------------------------------------------------------------------

// Includes

//-----------------------------------------------------------------------------

#include "..\Headers\SYS\_DEFS.H" // System type definitions and all includes

//----------------------------------------------------------------------------

/\* ---------------------------------------------------------------------------

Description :

Simultaneously writes and reads an 8-bit on the CC2420 SPIO interface.

Assumes that slave select line NSSMD0 is enabled before entry - avoids unecessary toggling the select signal

---------------------------------------------------------------------------- \*/

BYTE SPIO (BYTE TxByte)

{

SPI0DAT = TxByte; // Write byte to spi data register

while (!SPIF); // Wait for end of transfer

SPIF = 0; // Clear the SPI flag - to enable next transfer

return SPI0DAT; // Returning the byte captured on MISO

}

/\* ---------------------------------------------------------------------------

Description :

Reads a 16-bit register data value from a CC2420 Register Address

---------------------------------------------------------------------------- \*/

void WriteRegister(BYTE addr,WORD regval)

{

SPIO(addr); // Write register address

SPIO(regval >> 8); // Write Data MSByte

SPIO(regval); // Write Data LSByte

}

/\* ---------------------------------------------------------------------------

Description :

Reads a 16-bit register data value from a CC2420 Register Address

---------------------------------------------------------------------------- \*/

WORD ReadRegister(BYTE addr)

{

WORD regval;

SPIO(REGREAD | addr); // Enable bit6 and send the register address.

regval = ((SPIO(CC2420\_SNOP)<<8)+ SPIO(CC2420\_SNOP)); // Read MSByte + LSByte.

return regval; // Return the 16-bit register value.

}

/\* ---------------------------------------------------------------------------

Description :

Writes a 16-bit address value at a specified CC2420 RAM Address

---------------------------------------------------------------------------- \*/

WORD ReadRAM(WORD RAM\_Addr)

{

BYTE LSB, MSB;

SFRPAGE = SPI0\_PAGE; // Restore SFRs so that SPI data can be read

NSSMD0 = 0; // Enable CC2420 Chip Select

SPIO( RAMREADWRITE | (RAM\_Addr & 0x7F)); // Send RAM address LSB

SPIO(((RAM\_Addr >>1 ) & 0xC0) |RAMREAD); // Send RAM address LSB

LSB = SPIO(CC2420\_SNOP); // Read Data LSByte

MSB = SPIO(CC2420\_SNOP);

NSSMD0 = 1; // Disable Slave Select

return ((MSB << 8) | LSB); // Return 16-bit value

}

/\* ---------------------------------------------------------------------------

Author : Philip Cass

Last Modified : 12th March,2009 Created : 15th April, 2010

Name : void LongAddress (BYTE\* long\_address, BYTE write)

Inputs : BYTE\* - length 8 byte for IEEE long address

BYTE - True if address is to be written from array into

the cc2420; False if the opposite

Outputs : None

Description :

Writes an IEEE 64-bit address either from the cc2420 to an array, or

vice versa

---------------------------------------------------------------------------- \*/

void LongAddress(BYTE\* long\_address, BYTE write)

{

BYTE i,output;

SFRPAGE = SPI0\_PAGE; // Restore SFRs so that SPI data can be read

NSSMD0 = 0; // Enable CC2420 Chip Select

SPIO( RAMREADWRITE | (CC2420RAM\_IEEEADDR & 0x7F)); // Send RAM address LSB

SPIO(((CC2420RAM\_IEEEADDR >>1 ) & 0xC0) | (write ? RAMWRITE : RAMREAD)); // Send RAM address LSB

for (i=0;i<8;i++){

output=SPIO(long\_address[i]);

if (!write) {

long\_address[i]=output;

}

}

NSSMD0 = 1; // Disable Slave Select

}

/\* ---------------------------------------------------------------------------

Description :

Writes a 16-bit address value at a specified CC2420 RAM Address

---------------------------------------------------------------------------- \*/

void WriteRAM(WORD RAMdata,WORD RAM\_Addr)

{

SFRPAGE = SPI0\_PAGE; // Restore SFRs so that SPI data can be read

NSSMD0 = 0; // Enable CC2420 Chip Select

SPIO( RAMREADWRITE | (RAM\_Addr & 0x7F)); // Send RAM address LSB

SPIO(((RAM\_Addr >>1 ) & 0xC0) |RAMWRITE); // Send RAM address MSB

SPIO(RAMdata); // Write Data LSByte

SPIO(RAMdata >> 8); // Write Data MSByte

NSSMD0 = 1; // Disable Slave Select

}

/\* --------------------------------------------------------------------------- :

Description : Simple software delay loop

---------------------------------------------------------------------------- \*/

void Delay (unsigned int dely)

{

while (dely--); // Delay loop = Dely

}

### timings.h

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Author : Philip Cass

Modified : Philip Cass

Last Modified : 20th May, 2010 Created : 1st March,2010

Filename : timings.h

Target Hardware : Silicon Labs - C8051F120

Tool chain : KEIL C51 6.03 / KEIL EVAL C51

Version : 1.0.0

Description :

Constants and structs used for timing in the 802.15.4 network

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//Constants

#define BEACON\_ORDER 5 //Max of 14 = 2^B\_O times smallest possible superframe

#define BEACON\_ORDER\_ACTIVE 3 //Amount of the above in which the coordinator will listen

#if BEACON\_ORDER\_ACTIVE > BEACON\_ORDER

#undef BEACON\_ORDER\_ACTIVE

#define BEACON\_ORDER\_ACTIVE BEACON\_ORDER

#endif

#define SYMBOLS\_PER\_TICK 20 // At 3125Hz there's 20 symbols per tick

#define SLOTS\_PER\_SUPERFRAME 16

#define BASE\_SYMBOLS\_PER\_SLOT 60

#define BASE\_SUPERFRAME\_SIZE (SLOTS\_PER\_SUPERFRAME \* BASE\_SYMBOLS\_PER\_SLOT) //960 symbols = 480bytes/1000 = 15.36ms

#define BEACON\_INTERVAL\_TICKS ( (UINT32) (BASE\_SUPERFRAME\_SIZE / SYMBOLS\_PER\_TICK) << BEACON\_ORDER)

#define BEACON\_INTERVAL\_ACTIVE ( (UINT32) (BASE\_SUPERFRAME\_SIZE / SYMBOLS\_PER\_TICK) << BEACON\_ORDER\_ACTIVE)

//Most things here are given in symbols, not fractions of a second

#define POWER\_TO\_IDLE 63 //Actually 62.5

#define IDLE\_TO\_ACTIVE 12

#define FUDGE\_FACTOR 20 //Amount to pad the numbers by

// Radio will start power-up this long before beacon

#define POWER\_ON\_TIME ((POWER\_TO\_IDLE + IDLE\_TO\_ACTIVE + FUDGE\_FACTOR) / SYMBOLS\_PER\_TICK + 1)

// Radio will switch off if at least this time before beacon

#define POWER\_OFF\_IF\_TIME\_REMAINING ((POWER\_ON\_TIME + 100) / SYMBOLS\_PER\_TICK)

// Don't attempt to send packet if less than this time remaining

#define STOP\_SENDING\_PACKETS (300 / SYMBOLS\_PER\_TICK) // Frame size of 254 symbols + preamble + safety factor

// Time it takes a node to process a beacon - the beacon timer will be decremented by this value

// This is time to receive + time to process the ISR

#define BEACON\_PROCESSING\_DELAY ( 254 / SYMBOLS\_PER\_TICK)

typedef struct{

UINT32 current\_value; // Has to be this large as it can be 16,384 standard superframe

UINT32 reload\_value; // sizes between beacons, or 4 minutes and 6.41536 seconds

BYTE triggered; // Whoever's interested in this should reset it when they see it

}Timer;

### CO8051F121TB\_DEFS.H

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Author : Frank Greig

Modified : Philip Cass

Date : 20th May, 2010

Filename : CO8051F121TB\_DEFS.H

Target Hardware : Silicon Labs - C8051F121

Tool chain : KEIL C51 6.03 / KEIL EVAL C51

Version : 1.0.0

Description :

Global declaration file for C8051F121 MCU >> CC2420 Interface

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "..\Headers\C8051F120.H" // SFR declarations

#include "..\Headers\C8051F120sfr16.H" // SFR declarations

//-----------------------------------------------------------------------------

// Global CONSTANTS

//-----------------------------------------------------------------------------

//Declare this to use the faster internal clock

#define INTERNAL\_CLOCK FALSE

#if defined INTERNAL\_CLOCK

#else

#define INTERNAL\_CLOCK FALSE

#endif

#define BEACON\_PAYLOAD\_LENGTH 4

#define BEACON\_PAYLOAD "PGC\x01" // What the beacon transmits; what nodes look for

#define NUM\_TIMERS 2

#define TIMER\_LOOP\_VAR BYTE

#define T\_BEACON 0

#define T\_APPLICATION 1

#define SHOW\_RADIO TRUE // If true, use status LED to show when radio enabled

#if defined SHOW\_RADIO

#else

#define SHOW\_RADIO FALSE

#endif

// Outputting a lot of data on the serial \_WILL\_ cause timing issues!!

// It will cause them a lot faster at the slower speed

// But even on the faster clock, wordy debug output can affect the results

#define T3\_Overflow\_Rate 3125 // Timer3 Overflow frequency in Hz - 20 802.15.4 symbol periods per trigger

extern xdata Timer timers[NUM\_TIMERS];

// Set sysclock constant and UART speed based on clock choice

#if INTERNAL\_CLOCK

#define SYSCLK 24500000 // Internal frequency in Hz

#define BAUDRATE 57600 // Baud rate of UART in bps

#else

#define SYSCLK 8000000 // External Xtalfrequency in Hz

#define BAUDRATE 38400 // Baud rate of UART in bps

#endif

#define P2LEDs P2 // Status LEDs on GPIO Port 2

sbit StatusLED0 = P2^0; // RED LEDs

sbit StatusLED1 = P2^1;

sbit StatusLED2 = P2^2;

sbit StatusLED3 = P2^3;

sbit StatusLED4 = P2^4;

sbit StatusLED5 = P2^5;

sbit StatusLED6 = P2^6;

sbit StatusLED7 = P2^7; // GREEN LED

#define P3Switches P3 // Pushswitches on Target Board

sbit SW3 = P3^3;

sbit SW2 = P3^2;

sbit SW1 = P3^1;

sbit SW0 = P3^0;

//-----------------------------------------------------------------------------

// SPI GPIO Assignments

//-----------------------------------------------------------------------------

unsigned char SPIO (unsigned char SPI\_byte); // Transfer a byte across the SPI link

// Pin Declarations for MCU >> CC2420 SPI Interface

//

// sbit MOSI = P0^4; // Master Out / Slave In (output)

// sbit MISO = P0^3; // Master In / Slave Out (input)

// sbit SCK = P0^2; // Serial Clock (output)

sbit CSn = P0^5; // NSS Slave Chip select - set this to open drain input with weak pull-up

sbit VREGen = P1^7; // Input to CC2420

sbit RESETn = P1^6; // Input to CC2420

sbit CCA = P1^5; // Output from CC2420

sbit FIFO = P1^4; // In/Out - put (see for instance page 40 of CC2420 datasheet version 1.3!)

sbit FIFOP = P1^2; // Output from CC2420 to MCU INT0

sbit SFD = P1^0; // Output from CC2420

//-----------------------------------------------------------------------------

// Function PROTOTYPES

//-----------------------------------------------------------------------------

void SYSCLK\_Init (void);

void PORT\_Init (void);

void UART0\_Init (void);

void ADC0\_Init (void);

void SPI0\_Init(void);

void Timer3\_Init(void);

void FIFOP\_ISR (void); // External INT1 on P0.1

void Timer3\_ISR (void);

//-----------------------------------------------------------------------------

// End of file CO8051F121TB\_DEFS.H

### CO2420RF\_DEFS.H

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Author : Frank Greig

Modified : Philip Cass

Date : 20th May, 2010

Filename : CC2420RF\_DEFS.H

Target Hardware : Silicon Labs - C8051F120 Target Board

Tool chain : KEIL C51 7.05 / KEIL C51

Version : 1.0.0

Description :

Global declaration file for C8051F121 MCU >> CC2420 - I/O Target Board Interface

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//-----------------------------------------------------------------------------

// Global CONSTANTS

//-----------------------------------------------------------------------------

// Read and Write Defines for the 8-bit and 16 bit commands

#define REGREAD 0x40 // MSB 8 and 7 contain : 01 >> read

#define REGWRITE 0x00 // MSB 8 and 7 contain : 00 >> write

#define RAMREADWRITE 0x80 // APart 8 contain : 1 >> read/write RAM

#define RAMREAD 0x20 // Bpart 8 contain : 1 >> read RAM

#define RAMWRITE 0x00 // BPart 8 contain : 0 >> read/write RAM

#define PKT\_FOOTER\_LEN 2 // FCS = RSSI + CRC = 2

#define MAX\_PACKET\_SIZE 127 // 1x6 + 8x9 Data bytes + FCS

#define Query\_PACKET\_SIZE 6 // Packet data bytes only

#define Response\_PACKET\_SIZE 9 // Packet data bytes only

// Frame control values

#define FC\_TYPE\_BEACON 0x0

#define FC\_TYPE\_DATA 0x1

#define FC\_TYPE\_ACK 0x2

#define FC\_TYPE\_COMMAND 0x3

#define FC\_SECURITY\_ON 0x1 << 3

#define FC\_FRAME\_PENDING 0x1 << 4

#define FC\_ACK\_REQUESTED 0x1 << 5

#define FC\_INTRA\_PAN 0x1 << 6

#define FC\_ADDRESSING\_NONE 0x0

#define FC\_ADDRESSING\_SHORT 0x2

#define FC\_ADDRESSING\_LONG 0x3

#define FC\_SOURCE\_SHIFT 6

#define FC\_DEST\_SHIFT 2

// Beacon Superframe spec

#define B\_SF\_BEACON\_ORDER\_SHIFT 0

#define B\_SF\_SFRAME\_ORDER\_SHIFT 4

#define B\_SF\_LAST\_CAP\_SHIFT 0

#define B\_SF\_BATTERY\_EXTEND 0x1 << 4

#define B\_SF\_PAN\_COORDINATOR 0x1 << 6

#define B\_SF\_PERMIT\_ASSOCIATION 0x1 << 7

// GTS Spec field

#define B\_GTS\_COUNT\_SHIFT 0

#define B\_GTS\_PERMIT\_REQUESTS 0x1 << 7

// Pending transfers field

#define B\_PEND\_SHORT\_SHIFT 0

#define B\_PEND\_LONG\_SHIFT 4

//-----------------------------------------------------------------------------

// Global CONSTANTS

//-----------------------------------------------------------------------------

// Received Packet Status

#define PKT\_OK 0x00

#define BAD\_PKT 0x02

#define BAD\_CRC 0x03

#define PKT\_DONE 0x04

//-----------------------------------------------------------------------------

// Global VARIABLES

//-----------------------------------------------------------------------------

// These are declared here but defined in the PHY\_Comms file

// This prevents multiple definitions if this file is included more that once

extern xdata BYTE IEEE\_address[8];

//---------------------------------------------------------------------------

// Packet format

typedef struct{

BYTE DataLength; //number of Data bytes to send (not including RSSI/CRC)

BYTE \*PacketData; //

}Packet;

typedef struct{

// size of the source/destination fields (0, 2 or 8)

BYTE sourceLength;

BYTE destLength;

// only valid is the relevant length is > 0

UINT16 sourcePanId;

UINT16 destPanId;

// only valid up to the length specified

BYTE \*sourceAddress;

BYTE \*destAddress;

BYTE sequenceNum;

BYTE beaconOrder;

BYTE beaconActiveOrder;

BYTE payloadLength;

BYTE \*payload;

}beaconPacket;

//---------------------------------------------------------------------------

BYTE SPIO (BYTE TxByte);

void WriteStrobe(BYTE s);

BYTE ReadStatus(); // Get status byte from CC2420

WORD ReadRegister(BYTE addr); // Read CC2420 16-bit register

void WriteRegister(BYTE addr,WORD v);

WORD ReadRAM(WORD RAM\_Addr);

void WriteRAM(WORD RAMdata,WORD RAM\_Addr);

void LongAddress(BYTE\* long\_address, BYTE write);

void RxON(void);

void RxOFF(void);

void Initialise\_CC2420(); // Initialises the CC2420

void LoadPacket(Packet \*pkt);

void TransmitPacket(Packet \*pkt);

BYTE ReadFIFO\_Packet(Packet \*pkt); // Read a packet from RXFIFO

void SetRFChannel(BYTE channel);

void SetTxStrength(BYTE strength);

void FlushRxFIFO(void);

void FlushTxFIFO(void);

void Delay (unsigned int dely);

//---------------------------------------------------------------------------

## 802.15.4 network protocol – coordinator

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Author : Philip Cass

Last Modified : 26th May,2010 Created : 18th February, 2009

File : TxPackets-macpkts.c

Target Hardware : Silicon Labs - C8051F121 - I/O Target Board

Tool chain : KEIL C51 V7.05

Version : 1.0.0

Description :

Acts as a coordinator node in a 802.15.4 beacon network

Transmits packets at intervals dictated by values in timings.h

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//-----------------------------------------------------------------------------

// Includes

//-----------------------------------------------------------------------------

#include "..\Headers\SYS\_DEFS.H" // System type definitions and all includes

#include <stdio.h>

#include "..\protothreads\pt.h"

//-----------------------------------------------------------------------------

// Global CONSTANTS

//-----------------------------------------------------------------------------

#define Qpkt\_DataLength 17 // Payload data length

#define PacketRate T3\_Overflow\_Rate/50 // Interval between Qpkts : PacketRate x 50 Millseconds

xdata Timer timers[NUM\_TIMERS];

static struct pt pt\_beacon,pt\_manage,pt\_app;

#define CLOCK\_CORRECTION 129

//-----------------------------------------------------------------------------

// Global VARIABLES

//-----------------------------------------------------------------------------

BYTE DSN,BSN;

WORD PanId,ShortAddress;

char clock\_correction = CLOCK\_CORRECTION;

void Initialise\_timers (void)

{

TIMER\_LOOP\_VAR i;

for (i=0;i < NUM\_TIMERS;i++)

{

timers[i].current\_value=0;

timers[i].reload\_value=0;

timers[i].triggered=FALSE;

}

}

// This handles the superframe timing and so the general board state (when it can send/receive etc.)

static int beacon\_thread(struct pt \*pt){

static Packet transmitPacket; // This is the only thread which transmits

static xdata BYTE TxBuffer[MAX\_PACKET\_SIZE-PKT\_FOOTER\_LEN]; // Footer is auto-generated

BYTE i;

BYTE count;

PT\_BEGIN(pt);

transmitPacket.DataLength=0;

transmitPacket.PacketData=TxBuffer;

PT\_YIELD\_UNTIL(pt,timers[T\_BEACON].triggered); // Delay starting

timers[T\_BEACON].triggered=FALSE;

#if SHOW\_RADIO

StatusLED7=0;

#endif

while (1)

{

//Start with radio on, beacon ready to be sent

transmitPacket.PacketData[0] = FC\_TYPE\_BEACON;

transmitPacket.PacketData[1] = (FC\_ADDRESSING\_NONE << FC\_DEST\_SHIFT) | (FC\_ADDRESSING\_SHORT << FC\_SOURCE\_SHIFT);

transmitPacket.PacketData[2] = BSN;

transmitPacket.PacketData[3] = PanId;

transmitPacket.PacketData[4] = PanId >> 8;

transmitPacket.PacketData[5] = ShortAddress;

transmitPacket.PacketData[6] = ShortAddress >> 8;

transmitPacket.PacketData[7] = (BEACON\_ORDER << B\_SF\_BEACON\_ORDER\_SHIFT) | (BEACON\_ORDER\_ACTIVE << B\_SF\_SFRAME\_ORDER\_SHIFT);

transmitPacket.PacketData[8] = (0xF << B\_SF\_LAST\_CAP\_SHIFT) | B\_SF\_PAN\_COORDINATOR | B\_SF\_PERMIT\_ASSOCIATION; // 0xF = all slots in CAP

transmitPacket.PacketData[9] = (0x00 << B\_GTS\_COUNT\_SHIFT );

transmitPacket.PacketData[10] = (0x00 << B\_PEND\_SHORT\_SHIFT ) | (0x0 << B\_PEND\_LONG\_SHIFT);

count=11;

for (i=0; i < BEACON\_PAYLOAD\_LENGTH; i++){

transmitPacket.PacketData[count] = BEACON\_PAYLOAD[i];

count++;

}

transmitPacket.DataLength=count;

LoadPacket(&transmitPacket);

NSSMD0 = 0; // Enable CC2420

SPIO(CC2420\_STXON); // Send immediately WITHOUT CCA

NSSMD0 = 1; // Disable CC2420

BSN++;

if (BEACON\_INTERVAL\_TICKS != BEACON\_INTERVAL\_ACTIVE) {

PT\_WAIT\_UNTIL(pt,timers[T\_BEACON].current\_value < (BEACON\_INTERVAL\_TICKS - BEACON\_INTERVAL\_ACTIVE)); // Wait until receive window closes

if (timers[T\_BEACON].current\_value > POWER\_OFF\_IF\_TIME\_REMAINING && !timers[T\_BEACON].triggered) {

SFRPAGE = SPI0\_PAGE;

NSSMD0 = 0;

SPIO(CC2420\_SXOSCOFF); // Turn the oscillator off

#if SHOW\_RADIO

StatusLED7=1;

#endif

NSSMD0 = 1;

PT\_WAIT\_UNTIL(pt,(timers[T\_BEACON].current\_value < POWER\_ON\_TIME) || timers[T\_BEACON].triggered);

#if SHOW\_RADIO

StatusLED7=0;

#endif

SFRPAGE = SPI0\_PAGE;

NSSMD0 = 0;

SPIO(CC2420\_SXOSCON); // Turn the oscillator on

do{ // Wait for the XTAL oscillator to stablilise

} while (!( SPIO(CC2420\_SNOP) & (BM(CC2420\_XOSC16M\_STABLE)) ));

NSSMD0 = 1;

}

if (timers[T\_BEACON].triggered) {

StatusLED6=1;

}

}

PT\_WAIT\_UNTIL(pt,timers[T\_BEACON].triggered);

timers[T\_BEACON].triggered=FALSE;

}

PT\_END(pt);

}

//This handles any command frames that might be received i.e. association/dissociation etc.

static int management\_thread(struct pt \*pt){

PT\_BEGIN(pt);

DSN = 45; // Default sequence number

BSN = 127; // Default beacon sequence number

PanId = 0xDEAD;

ShortAddress = 0xBEEF;

SFRPAGE = SPI0\_PAGE;

WriteRAM(PanId,CC2420RAM\_PANID); // Initialse RAM with AR addresses

WriteRAM(ShortAddress,CC2420RAM\_SHORTADDR); // Must be done with OSC running

while (1)

{

PT\_YIELD(pt);

}

PT\_END(pt);

}

//Application as defined by the user - does whatever the user wants

static int application\_thread(struct pt \*pt){

PT\_BEGIN(pt);

while (1)

{

PT\_YIELD\_UNTIL(pt,timers[T\_APPLICATION].triggered);

timers[T\_APPLICATION].triggered=FALSE;

SFRPAGE = CONFIG\_PAGE;

StatusLED0 = ~StatusLED0;

}

PT\_END(pt);

}

//-----------------------------------------------------------------------------

// MAIN Routine

//-----------------------------------------------------------------------------

void main (void)

{

UINT32 a,b,c,d,e;

SYSCLK\_Init(); // Initialize SYSCLK system clock

PORT\_Init (); // Initialize XBAR crossbar and GPIO

UART0\_Init (); // Initialize UART0 Databits = 8, Stopbits =1

SPI0\_Init(); // Initialize SPIO -> CC2420 Interface

Timer3\_Init(); // Generate Timer3 based interrupts intervals.

Initialise\_CC2420(); // Init radio receiver

NSSMD0 = 0; // Enable CC2420 Chip Select

WriteRegister(CC2420\_MDMCTRL0,ReadRegister(CC2420\_MDMCTRL0) | 0x10); // Add "PAN coordinator" role

NSSMD0 = 1; // Disable CC2420 Chip Select

SetRFChannel(0x0C); // Set Channel

SetTxStrength(15);

a=SYMBOLS\_PER\_TICK;

b=BASE\_SUPERFRAME\_SIZE;

c=(1024 \* (UINT32) 48);

d=BEACON\_INTERVAL\_TICKS;

e=BEACON\_INTERVAL\_ACTIVE;

Initialise\_timers();

EA = 1; // Enable Global interrupts

EIE2 |= 0x01; // Enable Timer3 interrupts

SFRPAGE = UART0\_PAGE; // Restore SFRs so that serial data can be written

PT\_INIT(&pt\_beacon);

PT\_INIT(&pt\_manage);

PT\_INIT(&pt\_app);

timers[T\_APPLICATION].current\_value=3125;

timers[T\_APPLICATION].reload\_value=3125;

timers[T\_BEACON].current\_value=3125;

timers[T\_BEACON].reload\_value=BEACON\_INTERVAL\_TICKS;

while(1) {

beacon\_thread(&pt\_beacon);

management\_thread(&pt\_manage);

application\_thread(&pt\_app);

}

} // end main

//-----------------------------------------------------------------------------

// Timer3\_ISR

//-----------------------------------------------------------------------------

void Timer3\_ISR (void) interrupt 14

{

TIMER\_LOOP\_VAR i;

clock\_correction--;

SFRPAGE = TMR3\_PAGE;

TMR3CN &= ~0x80; // Reset TF3 flag

// if (clock\_correction==0) {

// clock\_correction=CLOCK\_CORRECTION;

// } else {

for (i=0;i < NUM\_TIMERS;i++)

{

if (timers[i].current\_value != 0)

{

timers[i].current\_value = timers[i].current\_value-1;

if (timers[i].current\_value==0)

{

timers[i].triggered=TRUE;

timers[i].current\_value = timers[i].reload\_value;

}

}

}

// }

}

## 802.15.4 network protocol – sensor node

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Author : Philip Cass

Last Modified : 26th May,2010 Created : 18th February, 2009

Target Hardware : Silicon Labs - C8051F121 - I/O Target Board

Tool chain : KEIL C51 V7.05

Version : 1.0.0

Description :

Received beacon frames from coordinator

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//-----------------------------------------------------------------------------

// Includes

//-----------------------------------------------------------------------------

#include "..\Headers\SYS\_DEFS.H" // System type definitions and all includes

#include <stdio.h>

#include "..\protothreads\pt.h"

//-----------------------------------------------------------------------------

// Global CONSTANTS

//-----------------------------------------------------------------------------

xdata Timer timers[NUM\_TIMERS];

static struct pt pt\_beacon,pt\_manage,pt\_app;

sbit status = P3^5;

//-----------------------------------------------------------------------------

// Global VARIABLES

//-----------------------------------------------------------------------------

BYTE DSN,BSN;

WORD PanId,ShortAddress;

BYTE beacon\_active;

Packet appTransmitPacket;

xdata BYTE appTxBuffer[MAX\_PACKET\_SIZE-PKT\_FOOTER\_LEN]; // Footer is auto-generated

UINT32 beacon\_stop\_transmitting;

beaconPacket beacon\_packet;

xdata BYTE beacon\_src[8];

xdata BYTE beacon\_dst[8];

xdata BYTE beacon\_payload[127];

void Initialise\_timers (void)

{

TIMER\_LOOP\_VAR i;

for (i=0;i < NUM\_TIMERS;i++)

{

timers[i].current\_value=0;

timers[i].reload\_value=0;

timers[i].triggered=FALSE;

}

}

// This handles the superframe timing and so the general board state (when it can send/receive etc.)

static int beacon\_thread(struct pt \*pt){

//static Packet transmitPacket; // This is the only thread which transmits

//static xdata BYTE TxBuffer[MAX\_PACKET\_SIZE-PKT\_FOOTER\_LEN]; // Footer is auto-generated

BYTE intStatus;

PT\_BEGIN(pt);

#if SHOW\_RADIO

StatusLED7=0;

#endif

while (1)

{

//Start with radio on, beacon ready to be received

PT\_WAIT\_UNTIL(pt,beacon\_active);

beacon\_active=0;

//appTransmitPacket.DataLength=8;

//We are now in the active period where we can send data if we have some

if (appTransmitPacket.DataLength > 0) {

// Load frame into output FIFO

LoadPacket(&appTransmitPacket);

// Send packet if we have time

while (timers[T\_BEACON].current\_value > beacon\_stop\_transmitting) {

intStatus = EX0; //save current status of int0

EX0 = 0; //regardless, turn int0 off

NSSMD0 = 0; // Enable Slave Select

SPIO(CC2420\_STXONCCA); // Send if clear

if (SPIO(CC2420\_SNOP) & (BM(CC2420\_TX\_ACTIVE))) {

EX0 = intStatus;

NSSMD0 = 1; // Disable CC2420

appTransmitPacket.DataLength = 0; // Indicate the packet has been sent

break;

}

// If we're still here, we didn't send... yeild the protothread to let something else get to run

EX0 = intStatus;

NSSMD0 = 1; // Disable CC2420

PT\_YIELD(pt);

} // Either packet sent, or we ran out of time

}

//Shut down the radio if we have time to do so

if (timers[T\_BEACON].current\_value > POWER\_OFF\_IF\_TIME\_REMAINING && !timers[T\_BEACON].triggered) {

intStatus = EX0; //save current status of int0

EX0 = 0; //regardless, turn int0 off

SFRPAGE = SPI0\_PAGE;

NSSMD0 = 0;

SPIO(CC2420\_SRFOFF); // Turn radio off

NSSMD0 = 1;

EX0 = intStatus;

PT\_YIELD(pt); // Chance to deal with any pending frames

intStatus = EX0; //save current status of int0

EX0 = 0; //regardless, turn int0 off

SFRPAGE = SPI0\_PAGE;

NSSMD0 = 0;

SPIO(CC2420\_SXOSCOFF); // Turn the oscillator off

#if SHOW\_RADIO

StatusLED7=1;

#endif

NSSMD0 = 1;

EX0 = intStatus;

PT\_WAIT\_UNTIL(pt,(timers[T\_BEACON].current\_value < POWER\_ON\_TIME) || timers[T\_BEACON].triggered);

#if SHOW\_RADIO

StatusLED7=0;

#endif

intStatus = EX0; //save current status of int0

EX0 = 0; //regardless, turn int0 off

SFRPAGE = SPI0\_PAGE;

NSSMD0 = 0;

SPIO(CC2420\_SXOSCON); // Turn the oscillator on

do{ // Wait for the XTAL oscillator to stablilise

} while (!( SPIO(CC2420\_SNOP) & (BM(CC2420\_XOSC16M\_STABLE)) ));

NSSMD0 = 1;

EX0 = intStatus;

}

timers[T\_BEACON].current\_value=0;

if (timers[T\_BEACON].triggered) {

StatusLED6=~StatusLED6;

timers[T\_BEACON].triggered=0;

// If this happens, we didn't turn the radio on in time

}

} // end of infinite loop

PT\_END(pt);

}

//This handles any command frames that might be received i.e. association/dissociation etc.

static int management\_thread(struct pt \*pt){

PT\_BEGIN(pt);

DSN = 45; // Default sequence number

BSN = 127; // Default beacon sequence number

beacon\_active=0;

appTransmitPacket.DataLength = 0;

beacon\_stop\_transmitting = BEACON\_INTERVAL\_TICKS - BEACON\_INTERVAL\_ACTIVE + STOP\_SENDING\_PACKETS;

PanId = 0xDEAD;

ShortAddress = 0xBEEF;

SFRPAGE = SPI0\_PAGE;

WriteRAM(PanId,CC2420RAM\_PANID); // Initialse RAM with AR addresses

WriteRAM(ShortAddress,CC2420RAM\_SHORTADDR); // Must be done with OSC running

NSSMD0 = 0;

SPIO(CC2420\_SRXON);

NSSMD0 = 1;

while (1)

{

PT\_YIELD(pt);

}

PT\_END(pt);

}

//Application as defined by the user - does whatever the user wants

static int application\_thread(struct pt \*pt){

PT\_BEGIN(pt);

while (1)

{

PT\_YIELD\_UNTIL(pt,timers[T\_APPLICATION].triggered);

timers[T\_APPLICATION].triggered=FALSE;

SFRPAGE = CONFIG\_PAGE;

StatusLED0 = ~StatusLED0;

}

PT\_END(pt);

}

//-----------------------------------------------------------------------------

// MAIN Routine

//-----------------------------------------------------------------------------

void main (void)

{

SYSCLK\_Init(); // Initialize SYSCLK system clock

PORT\_Init (); // Initialize XBAR crossbar and GPIO

UART0\_Init (); // Initialize UART0 Databits = 8, Stopbits =1

SPI0\_Init(); // Initialize SPIO -> CC2420 Interface

Timer3\_Init(); // Generate Timer3 based interrupts intervals.

beacon\_packet.sourceAddress = beacon\_src;

beacon\_packet.destAddress = beacon\_dst;

beacon\_packet.payload = beacon\_payload;

appTransmitPacket.DataLength=0;

appTransmitPacket.PacketData=appTxBuffer;

appTransmitPacket.PacketData[0]=FC\_TYPE\_DATA | FC\_ACK\_REQUESTED;

appTransmitPacket.PacketData[1]=(FC\_ADDRESSING\_NONE << FC\_DEST\_SHIFT) | (FC\_ADDRESSING\_SHORT << FC\_SOURCE\_SHIFT);

appTransmitPacket.PacketData[2]=0x01;

appTransmitPacket.PacketData[3]=0xAD;

appTransmitPacket.PacketData[4]=0xDE;

appTransmitPacket.PacketData[5]=0xEF;

appTransmitPacket.PacketData[6]=0xBE;

appTransmitPacket.PacketData[7]=0x65;

// appTransmitPacket.DataLength=8;

Initialise\_CC2420(); // Init radio receiver

NSSMD0 = 0; // Enable CC2420 Chip Select

WriteRegister(CC2420\_MDMCTRL0,ReadRegister(CC2420\_MDMCTRL0) | 0x10); // Add "PAN coordinator" role

NSSMD0 = 1; // Disable CC2420 Chip Select

SetRFChannel(0x0C); // Set Channel

SetTxStrength(15);

Initialise\_timers();

EA = 1; // Enable Global interrupts

EIE2 |= 0x01; // Enable Timer3 interrupts

EX0=1; IT0=0; // Enable INT0 as edge 1->0 sensitive

PT\_INIT(&pt\_beacon);

PT\_INIT(&pt\_manage);

PT\_INIT(&pt\_app);

timers[T\_APPLICATION].current\_value=3125;

timers[T\_APPLICATION].reload\_value=3125;

timers[T\_BEACON].current\_value=0;

timers[T\_BEACON].reload\_value=0;

while(1) {

beacon\_thread(&pt\_beacon);

management\_thread(&pt\_manage);

application\_thread(&pt\_app);

StatusLED6 = FIFOP;

}

} // end main

void FIFOP\_ISR (void) interrupt 0 // External INT0 on P0\_1

{

UINT8 fcf1;

UINT8 fcf2;

UINT8 length;

UINT8 seq;

UINT8 i; // how far along the frame we are

UINT8 j; // loop var

UINT8 k; // byte we've just read in

// size of the source/destination fields (0, 2 or 8)

BYTE sourceLength;

BYTE destLength;

// only valid is the relevant length is > 0

UINT16 sourcePanId;

UINT16 destPanId;

// only valid up to the length specified

BYTE beacon\_src[8];

BYTE beacon\_dst[8];

SFRPAGE = SPI0\_PAGE;

NSSMD0 = 0; // Enable CC2420

if (FIFO==1) {

// FIFO is reverse polarity, so this means FIFO is low

// FIFO low and FIFOP high means overflow]

SPIO(CC2420\_SFLUSHRX); //Clear Rx FIFO buffer

SPIO(CC2420\_SFLUSHRX);

NSSMD0 = 1; // Disable CC2420

return;

}

StatusLED5 = ~StatusLED5; // Toggle LED0 on FIFOP Interrupt

SPIO( REGREAD | CC2420\_RXFIFO); // Send read FIFO command

length = SPIO(CC2420\_SNOP) & 0x7F; // 1st byte is the length

fcf1 = SPIO(CC2420\_SNOP); // 1st byte of frame control field

fcf2 = SPIO(CC2420\_SNOP); // 2nd byte

seq = SPIO(CC2420\_SNOP); //Then sequence number

i=3; // Bytes consumed

switch ((fcf2 >> FC\_DEST\_SHIFT) & 0x0F) {

case FC\_ADDRESSING\_LONG:

destLength=8;

break;

case FC\_ADDRESSING\_SHORT:

destLength=2;

break;

case FC\_ADDRESSING\_NONE:

default:

destLength=0;

}

if (destLength>0){

destPanId=SPIO(CC2420\_SNOP) && (SPIO(CC2420\_SNOP) >> 8);

i+=2;

}

for (j=0;j<destLength;j++){

beacon\_dst[j]=SPIO(CC2420\_SNOP);

}

i+=destLength;

switch ((fcf2 >> FC\_SOURCE\_SHIFT) & 0x0F) {

case FC\_ADDRESSING\_LONG:

sourceLength=8;

break;

case FC\_ADDRESSING\_SHORT:

sourceLength=2;

break;

case FC\_ADDRESSING\_NONE:

default:

sourceLength=0;

}

if (sourceLength>0){

sourcePanId=SPIO(CC2420\_SNOP) && (SPIO(CC2420\_SNOP) >> 8);

i+=2;

}

for (j=0;j<sourceLength;j++){

beacon\_src[j]=SPIO(CC2420\_SNOP);

}

i+=sourceLength;

switch (fcf1 & 0x07) {

case FC\_TYPE\_BEACON:

k = SPIO(CC2420\_SNOP);

beacon\_packet.beaconOrder= (k >> B\_SF\_BEACON\_ORDER\_SHIFT) & 0x0F;

beacon\_packet.beaconActiveOrder = (k >> B\_SF\_SFRAME\_ORDER\_SHIFT) & 0x0F;

SPIO(CC2420\_SNOP); // GTS spec byte

SPIO(CC2420\_SNOP); // Pending addresses count

i+=3;

j=i;

for (; i < (length-2); i++)

{

beacon\_packet.payload[i-j]=SPIO(CC2420\_SNOP);

}

SPIO(CC2420\_SNOP); // RSSI value;

k = SPIO(CC2420\_SNOP); // CRC

if (k&0x70) { // Packet OK

timers[T\_BEACON].current\_value = (UINT32) ((BASE\_SUPERFRAME\_SIZE / SYMBOLS\_PER\_TICK) << beacon\_packet.beaconOrder) - BEACON\_PROCESSING\_DELAY;

beacon\_stop\_transmitting = (UINT32) ((BASE\_SUPERFRAME\_SIZE / SYMBOLS\_PER\_TICK) << beacon\_packet.beaconOrder) - ((BASE\_SUPERFRAME\_SIZE / SYMBOLS\_PER\_TICK) << beacon\_packet.beaconActiveOrder) + STOP\_SENDING\_PACKETS;

beacon\_active=1;

}

break;

case FC\_TYPE\_DATA:

StatusLED3=0;

break;

case FC\_TYPE\_ACK:

StatusLED2=0;

break;

case FC\_TYPE\_COMMAND:

StatusLED1=0;

break;

default:

break;

}

for (; i < length; i++)

{

putchar(SPIO(CC2420\_SNOP));

}

NSSMD0 = 1;

}

//-----------------------------------------------------------------------------

// Timer3\_ISR

//-----------------------------------------------------------------------------

void Timer3\_ISR (void) interrupt 14

{

TIMER\_LOOP\_VAR i;

SFRPAGE = TMR3\_PAGE;

TMR3CN &= ~0x80; // Reset TF3 flag

for (i=0;i < NUM\_TIMERS;i++)

{

if (timers[i].current\_value != 0)

{

timers[i].current\_value = timers[i].current\_value-1;

if (timers[i].current\_value==0)

{

timers[i].triggered=TRUE;

timers[i].current\_value = timers[i].reload\_value;

}

}

}

}

// end of TxPackets-macpkts.c

# Appendix 2 Week 9 overview

# Appendix 3 Diary sheets