TESTING OF A NEW WIRELESS EMBEDDED BOARD

by

Murari Raghavan

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in the
Department of Electrical and Computer Engineering

Charlotte

2005

Approved by:

____
Dr. James M. Conrad

____
Dr. Ivan L. Howitt

____
Dr. Bharathkumar S. Joshi

© 2005
Murari Raghavan
ALL RIGHTS RESERVED

ABSTRACT

MURARI RAGHAVAN Testing of a new wireless embedded board. (Under the direction of DR. JAMES M. CONRAD)

The test phase plays a significant role before commercialization of every product. Currently numerous test methods have been proposed and practiced in the industry. However, most methods implemented require expensive tools and complex methodologies. This provides enough scope for further research to reduce the complexity of the test phase maintaining the efficiency of the test cycle.

In this thesis, a wireless embedded evaluation board consisting Atmel ATmega 128L (Microcontroller), Chipcon CC2420 (RF transceiver), and MAXIM DS2740 (Coulomb counter) is used as a platform for implementing the proposed test plan. The major contribution of the proposed methodology resolves in reduction of the complexity of the test. It is achieved by treating the evaluation board as an integrated system comprising individual function blocks interfaced together. The system is broken down into microcontroller module, RF module and coulomb counter module. The test cycle consists of two phases. During the first test phase each module is tested individually and the expected results are compared with the obtained result. During the second stage the system is tested as a whole for its complete functionality. This method proved to be simple and effective for testing a wireless board of reasonable complexity.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Dr. James M. Conrad. His constant encouragement and systematic approach has been a guiding factor in the successful completion of this thesis work. His structured approach to identifying and solving a problem has been a major factor in completing this thesis earlier. I would also like to express my gratitude to Dr. Ivan L. Howitt and Dr. Bharatkumar S. Joshi for serving on my thesis committee. I particularly want to thank Assad Ansari and Rajan Rai for their continuous encouragement and support throughout the course of this thesis work.

TABLE OF CONTENTS

LIST OF FIGURES VII

LIST OF TABLE IX

CHAPTER 1: INTRODUCTION 1

- 1.1 Background 2
- 1.2 Motivation 4
- 1.3 Description of the Proposed Thesis Work 5
- 1.4 Organization of Thesis 7

CHAPTER 2: HARDWARE 8

- 2.1 ATMEL ATmega 128L AVR Microcontroller 9
- 2.2 CHIPCON CC2420 RF Transceiver 10
- 2.3 MAXIM DS2740 Coulomb Counter 12
- 2.3.1 Current Measurement 14

CHAPTER 3: INTERFACING 15

- 3.1 Introduction to Serial Peripheral Interfacing (SPI) 15
- 3.2 CC2420 Data Transfer Mechanism 16
  - 3.2.1 CC2420 Interfacing Specifications 18
  - 3.2.2 Receive Mode 18
  - 3.2.3 Transmit mode 20
- 3.3 One-Wire Interface 21

CHAPTER 4: TESTING METHODOLOGY 25

- 4.1 Introduction 25
- 4.2 Power Circuit 28

4.2.1 Testing ATmega 128L Power Supply 29
4.2.2 Testing CC2420 Power Supply 30
4.2.3 Testing DS2740 power supply 31
4.3 Crystal Oscillator 33
4.3.1 Test for Crystal Oscillator connected to ATmega 128L 33
4.3.2 Test for Crystal Oscillator connected to CC2420 33
4.4 Reset Circuit 34
4.5 JTAG Interface 35
4.6 User I/O 36
4.6.1 Test for Push Button 36
4.6.2 Test for LED 37
4.6.3 Test for MSV5 37
4.6.4 Test for MSV5X2 37
4.6.5 Test for MSV12 38
4.7 UART Interface 39
4.8 Test for CC2420 39
4.9 Test for DS2740 40
4.10 Testing full functionality of the board 41
CHAPTER 5: CONCLUSION 44
REFERENCES 49
APPENDIX A: SCHEMATICS 51
APPENDIX B: CODES 55

LIST OF FIGURES

Figure 1.1 Design productivity gap [3, p6]. 2

Figure 1.2 Design and test cycle [3, p8] 3

Figure 2.1 Architecture of the wireless embedded board 8

Figure 2.2 Architecture of ATMEL ATmega 128L AVR Microcontroller [1, p3]. 10

Figure 2.3 Simplified block diagram of CC2420 [5, p16]. 12

Figure 2.4 Block diagram of DS2740 [7, p3]. 13

Figure 3.1 SPI interface between a CC2420 and an ATmega 128L 15

Figure 3.2 SPI Timing diagram [5, p26]. 17

Figure 3.3 Pin activities during receive [6, p34]. 19

Figure 3. 4 Example of pin activity when reading RXFIFO [6, p34]. 20

Figure 3.5 Pin activity example during transmit [6]. 21

Figure 3.7 One-Wire Interface between ATmega128L and MAXIM DS2740 22

Figure 4.3 Location of Power circuit for ATmega 128L 29

Figure 4.4 CC2420 reference design provided by CHIPCON support [6] 30

Figure 4.5 Location of Power circuit for CC2420 31

Figure 4.6 Reference design for Coulomb counter DS2740 provided by MAXIM [8]. 32

Figure 4.7 Location of Power circuit for DS2740 32

Figure 4.8 Location of external crystal oscillator for ATmega 128L and CC2420 34

Figure 4.10 Location of LEDs and switches 36

Figure 4.11 Location of headers MSV5, MSV5X2 and MSV12 38

Figure 4.12 Block diagram of the experimental setup for CC2420 test 40

Figure 4.13 Block diagram of the experimental setup for DS2740 test 41

Figure 4.14 Set up to test full functionality of the board. 43

Figure 5.1 ATmega 128L microcontroller module setup. 45

Figure 5.2 CC2420 module set up using CHIPCON CC2420DBK evaluation board...... 46

Figure 5.3 CC2420 module set up using CHIPCON CC2420EB evaluation board. 47

Figure 5.4 DS2740 Coulomb counter module setup. 47

Figure A.1 Microcontroller ATmega128L interfaced with Coulomb counter DS2740 ... 51

Figure A.2 RF Transceiver CC2420 interfaced with Coulomb Counter DS2740 52

Figure A. 3 Power Supply for ATmega128L and CC2420 with some headers on the board 53

Figure A.4 MAX3243 RS-232 line Driver/Receiver 54

LIST OF TABLE

Table 3.1 SPI Timing specifications [5, p26] 17

LIST OF ABBREVIATIONS

ADC	Analog to Digital converter
ATmega 128L	AVR ATmega 128 L microcontroller
BS	Boundary Scan
CC2420	CHIPCON CC2420 RF transceiver
CCA	Clear Channel Assessment
CMOS	Complementary Metal Oxide Semiconductor
CRC	Cyclic Redundancy Check
CS	Chip Select
DAC	Digital-to-Analog Converter
DQ	Data Input/Output
DS2740	MAXMIM DS2740 Coulomb Counter
EEPROM	Electrically Erasable Programmable Read only Memory
FIFO	First In First Out
I/O	Input/Output
IF	Intermediate Frequency
IS1	Current-Sense Input
ISM	Industrial Scientific Medical radio band
JTAG	Joint Test Action Group
LNA	Low Noise Amplifier
MPDU	MAC Protocol Data Unit
OVD	1-Wire Bus Speed Select
PA	Power Amplifier

PIO	Programmable I/O Pin
PWM	Pulse Width Modulation
RISC	Reduced Instruction Set Computer
RSSI	Received Signal Strength Indication
RTC	Real Time Counter
RXFIFO	Receive FIFO
SCLK	System Clock
SFD	Start of Frame Delimiter
SI	System In
SNS	Sense Resistor Input
SO	System Out
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
TXFIFO	Transmit FIFO
V_{DD}	Power-Supply Input (2.7V to 5.5V)
V_{SS}	Device Ground, Current-Sense Resistor Return

CHAPTER 1: INTRODUCTION

The advent of a simple, low-cost and extremely low-power protocol like IEEE 802.15.4 has been a major factor in redefining home automation. The rapid growth of wireless gadgets for home automation can be attributed to the development of standards like IEEE 802.15.4 [9]. At this juncture the industry analysts predict a greater penetration of wireless gadget in every sector of home automation [16]. Nevertheless, 802.15.4 is considered to have a far reaching potential in many other sectors apart from home automation [20].

A typical product manufacturing cycle encompasses initial design, prototyping, testing and production. Every stage in the product cycle has its own significance in the final quality of the products. Although every development process is carried out with strict quality standards, the test phase is given a high priority status to ensure this factor. It has been identified that the test phase takes almost 70% of the total IC manufacturing cycle time [7]. Furthermore, the “Rule of 10” indicates that testing a defective IC on a board is 10 times more expensive than testing it separately [4]. It was further studied and observed that a fault escaping the test phase contributes a major share in the overall time taken for repair and maintenance of a particular product, thereby reducing the overall productivity.

1.1 Background

According to "Moore's Law", the transistor density of integrated circuits doubles about every two years. This law has been proven valid for the past 40 years. Researchers and designers are actively involved in reducing the size of the integrated circuits rapidly to develop atomic sized electronic devices. Figure 1.1 shows the transistor occupancy in an integrated chip between 1981 and 2005 and the productivity gap. An integrated chip consists of 10-100 million logic transistors. It has become extremely difficult and expensive to test such complex circuits. To reduce the difficulty of testing at the end of the manufacturing cycle, the industries constantly emphasize the importance of verification and testing in the scope of designing and testing cycle of electronic testing.

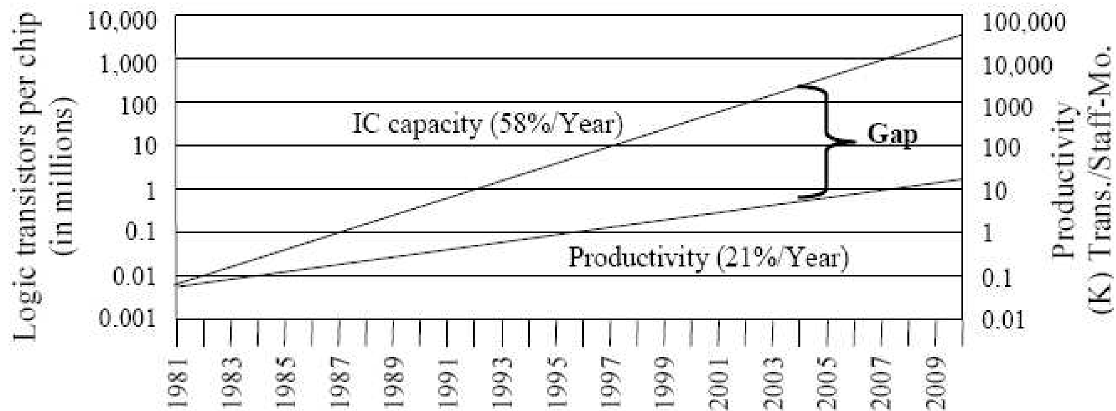


Figure 1.1 Design productivity gap [3, p6].

Figure 1.2 represents a typical design flow. The design starts from a specification, then flows to a behavioral or register transfer level (RTL). The design undergoes several synthesis steps before being transformed to the lower level of abstraction. At that time, the design is modeled by four different representations – behavioral, RTL, gate level, and layout. They are diverse representation of the same circuit. The mechanism used to check

the lower level representation for conformance with the higher one is called the design verification. There are two types of design verification – simulation and formal verification. The most important of the two methods is the simulation, which is primarily used to check that all blocks of the circuit perform the intended functions. Similar to the functional simulation at the chip level, a functional simulation is carried out at the board level. The system level testing involves testing the blocks of different modules present on the board.

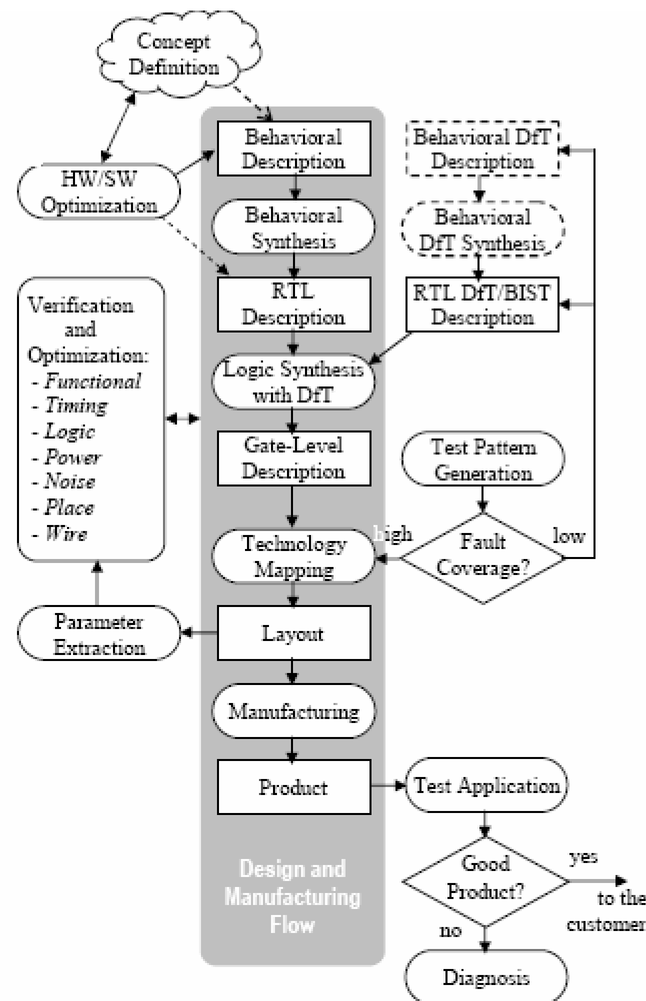


Figure 1.2 Design and test cycle [3, p8]

The PCB-level testing has been extremely simplified with the introduction of IEEE1149.1 Boundary Scan (BS) standard [12] in 1990. The Boundary Scan technique has reduced the test process to testing interconnect between chips. In a typical PCB there are several blocks of complex circuitries, such circuits are tied together to form an integrated system by glue logic. Though the Boundary Scan has simplified the PCB testing to a great extent, faults induced by glue logic still remains a major challenge. Another major issue concerning the researchers is the ability to perform at-speed testing. Apart from these issues, cross talk noise and other capacitance and inductance related effects due to reduced size also need to be addressed.

There are several techniques available in the market to test the interconnects between chips. Many test methods has been proposed utilizing on-board sensors [21], RF induction [5] and various other techniques. However, these methods demand complicated test fixtures and accurate analyzers to detect the faults.

1.2 Motivation

In the semiconductor industry, rapidly increasing design complexity and very tight time-to-market schedules have been prevalent for many years. Complex systems require complex test methodologies to identify a faulty area. With the increasing complexity of the electronic systems, testing has become a major area of research. The goal of this thesis is to consider a wireless embedded evaluation board of reasonable complexity, propose suitable testing methodology and implement the proposed test plan to identify the faulty area at the PCB-level. At this point we assume that the hardware has successfully undergone a chip level test.

The wireless evaluation board consists of ATMEL ATmega 128L AVR microcontroller, CHIPCON CC2420 RF transceiver and a MAXIM DS2740 coulomb counter. The test approach involved treating the microcontroller, RF transceiver and the coulomb counter as individual blocks. During the first stage of the test cycle the modules were tested for their individual functionality followed by testing the embedded system as a whole. Several simple executable images were written, compiled and downloaded to test different features of the microcontroller, RF transceiver and the coulomb counter. To ensure the efficiency of the code, the complied version was first downloaded on the evaluation board prescribed by the Atmel and CHICON. Further the programs were modified to suit the evaluation board under test. The following section presents an overall description of the research work.

1.3 Description of the Proposed Thesis Work

The main aim of the research project was to build a suitable platform to observe and analyze the power consumption of the microcontroller and the RF section during various stages of operation. A test process, test plan and test cases were created to validate this design.

A standard protocol like IEEE 802.15.4 has been an important factor in the growth of home automation. Any valuable observation of the power consumption of a system implementing IEEE 802.15.4 protocol would contribute towards building efficient networks. It was identified that the core infrastructure of the system should contain a microcontroller, an RF section and current sensors to measure the current consumed by the microcontroller and the RF section. Several microcontrollers were considered before selecting an AVR core microcontroller. ATmega 128L AVR microcontroller provided all

the necessary features to interface an RF section and current sensors. Testability was considered throughout the initial stages of design. The microcontroller is JTAG (IEEE 1149.1) compliant with boundary scan capabilities which provided an ideal platform for building test codes. A user friendly Integrated Development Environment (IDE) AVR Studio 4.0 was provided by Atmel support, where software modules can be written and debugged in windows platform. It also supports various AVR development kits like STK500, STK 501 which can be utilized to simulate the microcontroller modules. Chipcon CC2420 2.4GHz RF transceiver was identified as a suitable RF chip for its compatibility with IEEE 802.15.4 standard.

The Chipcon CC2420DBK evaluation kit was used to build a virtual environment to simulate the design and to test the software module. A coulomb counter evaluation board consisting MAXIM Dallas DS2740 high precision coulomb counter was chosen as an ideal platform to test the current sensor. The main modules of the design consists of ATmega 128L, CC2420 RF transceiver and DS2740 high precision coulomb counter. The test modules were identified during final stage of design cycle. A virtual environment using the prescribed evaluation boards were built to test the functional specification of the design. In order to test the initial prototype of the board, a test plan describing the test cases were developed. Module based test codes were written, compiled and downloaded onto the development kits to observe the behavior of the respective modules in the environment. To reduce the complexity of the test phase, the system was divided into 3 main modules; Microcontroller module, RF module and Coulomb counter module. A structured test plan was created. Test cases were identified and the respective

test codes were built and the executable images of the codes were tested in the virtual environment.

1.4 Organization of Thesis

This thesis report is divided into four major chapters. Chapter 2 of this thesis report discusses the hardware features of ATmega 128L microcontroller, CC2420 RF transceiver and DS2740 coulomb counter in separate sub sections. The architecture and the features of each module are also explained. Chapter 3 gives an overview of the interfacing techniques implemented followed by the description for Serial Peripheral Interfacing between the CC2420 and the ATmega 128L and One-Wire interface between the DS2740 and the ATmega 128L. SPI is explained with appropriate timing diagrams and pin configurations. The operation of the transceiver during various stages of operation is also discussed in detail in the subsections. One-Wire interfacing is described in detail during various stages of operation. A list of the important commands is included with a brief explanation of the functionality of each command. Chapter 4 presents the different phases involved in a product manufacturing cycle. The test agenda followed throughout the test phase is listed. This is followed by an introduction to the test methodologies implemented to test the wireless evaluation board. A brief summary of the research work undertaken and their conclusions are presented in Chapter 5. Appendix A illustrates the schematic of the wireless evaluation board. Appendix B consists of the source codes used during the test phase.

CHAPTER 2: HARDWARE

The embedded evaluation board under test consists of an ATMEL ATmega 128L AVR microcontroller, CHIPCON CC2420 RF transceiver and MAXIM DS2740 coulomb counter. It also contains switches and LEDs for providing various user input/output information [2]. The board is equipped with headers where many internal signals are available. Figure 2.1 gives an illustration of the architecture of the design.

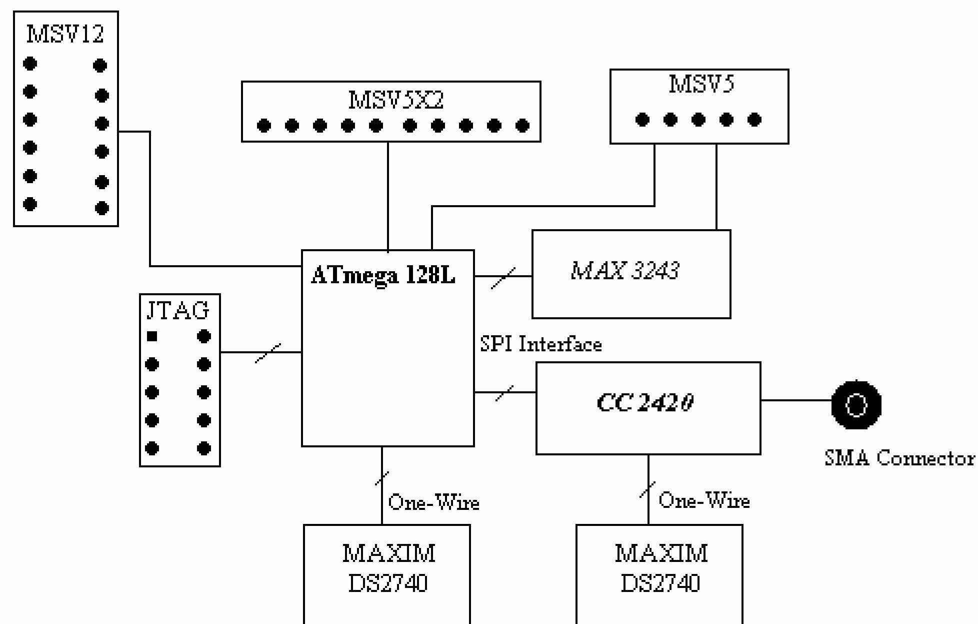


Figure 2.1 Architecture of the wireless embedded board

During normal operation in our research environment, two wireless embedded boards are used. One board acts as a transmitter, while the other as a receiver. When the boards

are powered, the CC2420 on the transmitting board transmits the value of the current drawn by the AVR microcontroller obtained by DS2740. The other board receives the value and displays it continuously on an attached PC running Hyperterminal. This setup provides an ideal platform to study and improve the test strategies that could be used to validate the evaluation board.

2.1 ATMEL ATmega 128L AVR Microcontroller

The ATMEL ATmega 128L AVR Microcontroller is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture [1]. The AVR core combines a rich instruction set with 32 general purpose working registers. Figure 2.2 illustrate the architecture of the AVR microcontroller.

The microcontroller features 128K byte of In-System programmable flash and 4K EEPROM with an endurance of 10,000 write/erase cycles. It supports up to 64 K of external memory and also provides an SPI interface for In-System programming and programming lock for security. It comes with a JTAG (IEEE Std 1149.1 compliant) interface. The Boundary Scan capabilities (JTAG standard) allows developers to observe the registers. Extensive on-chip debug support is also provided which reduces the debugging period of erroneous program.

The ATmega 128L also has several peripheral features like 8-bit/16-bit timers capable of operating in three different modes; presale, compare and capture modes are provided. It also consists of two 8-bit PWM channels, output compare modulator, 8-channel, 10-bit ADC, serial programmable USARTs, Master/slave SPI interface and many other essential features.

In our wireless embedded board only a limited number of the microcontroller features are used. With so many essential features unused, and considering the flexibility of the board architecture many peripheral devices can be attached to improve the overall functionality of the board in the future development.

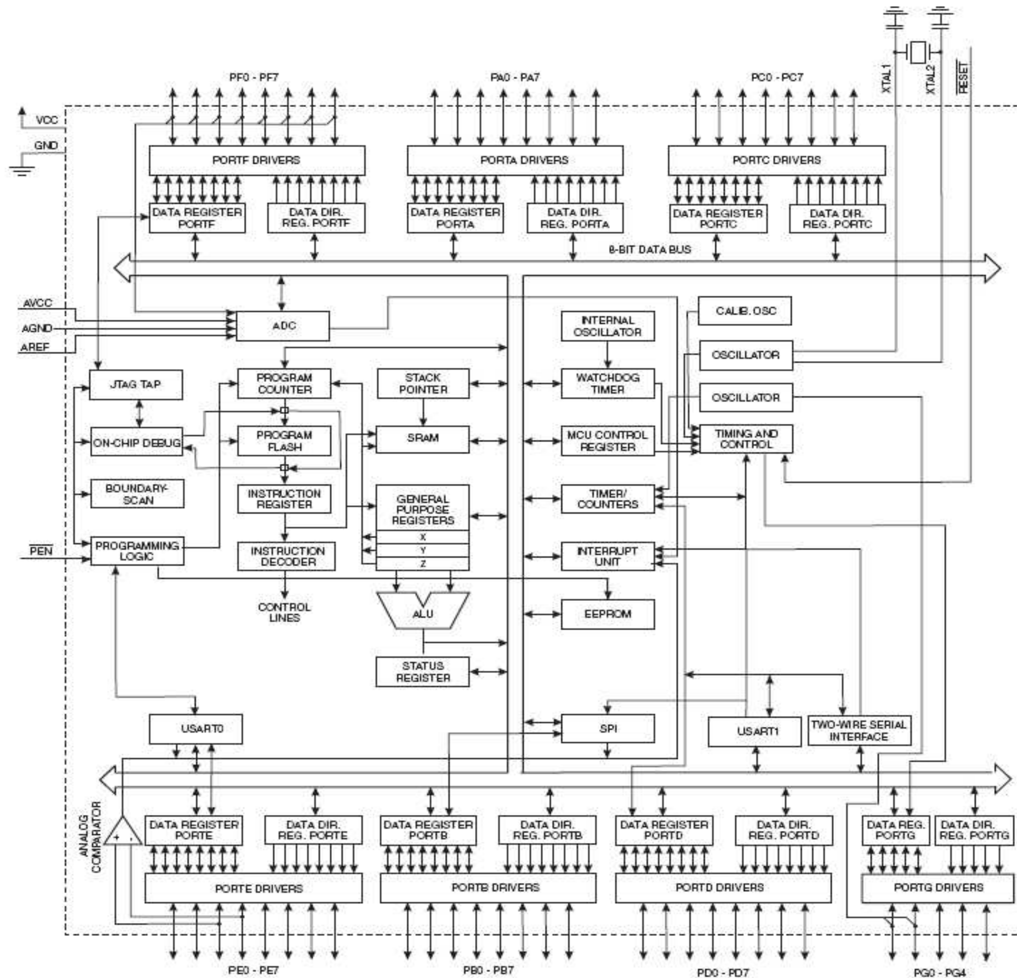


Figure 2.2 Architecture of ATMEL ATmega 128L AVR Microcontroller [1, p3].

2.2 CHIPCON CC2420 RF Transceiver

The CC2420 is a 2.4 GHz IEEE 802.15.4 compliant RF transceiver designed for low-power and low-voltage wireless applications [6]. The CC2420 includes a digital direct

sequence spread spectrum baseband modem providing a spreading gain of 9 dB and an effective data rate of 250 kbps. It is a low-cost, highly integrated solution for robust wireless communication in the 2.4 GHz unlicensed ISM band [10].

The CC2420 provides extensive hardware support for packet handling, data buffering, burst transmissions, data encryption, data authentication, clear channel assessment, link quality indication and packet timing information. These features reduce the load on the host controller and allow the CC2420 to interface with low-cost microcontrollers. The configuration interface and transmit/receive FIFO of the CC2420 are accessed via an SPI interface. In a typical application the CC2420 will be used together with a microcontroller and a few external passive components.

Figure 2.3 illustrates a simplified block diagram of the CC2420, which features a low-IF receiver. The received RF signal is amplified by the low-noise amplifier (LNA) and is down-converted in quadrature (I and Q) to the intermediate frequency (IF). At IF (2 MHz), the complex I/Q signal is filtered and amplified, and then digitized by the ADCs. Automatic gain control, final channel filtering, de-spreading, symbol correlation and byte synchronization are performed digitally.

The SFD pin goes high when a start of frame delimiter has been detected. The CC2420 buffers the received data in a 128 byte receive FIFO. The user may read the FIFO through an SPI interface [6]. The CRC byte is verified in hardware. RSSI and correlation values are appended to the frame. CCA is available on a pin in receive mode. Serial (unbuffered) data modes are also available for test purposes.

The CC2420 transmitter is based on direct up-conversion. The data is buffered in a 128 byte transmit FIFO (separate from the receive FIFO). The preamble and start of

frame delimiter are generated by hardware. Each symbol (4 bits) is spread using the IEEE 802.15.4 spreading sequence to 32 chips and output to the digital-to-analog converters (DACs). An analog low-pass filter passes the signal to the quadrature (I and Q) up-conversion mixers. The RF signal is amplified in the power amplifier (PA) and fed to the antenna.

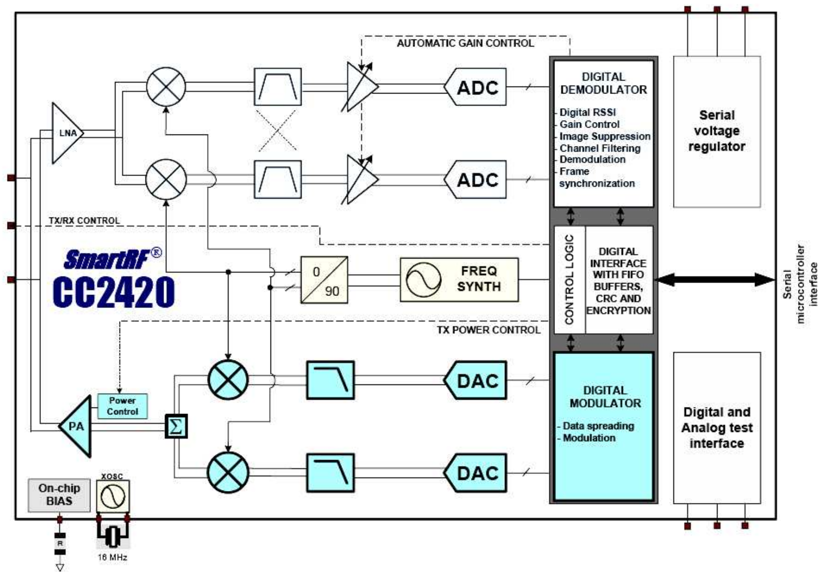


Figure 2.3 Simplified block diagram of CC2420 [5, p16].

2.3 MAXIM DS2740 Coulomb Counter

The DS2740 provides high-precision current-flow measurement data to support battery-capacity monitoring in cost-sensitive applications [8]. Current is measured bi-directionally over a dynamic range of 15bits (DS2740U) or 13 bits (DS2740BU), with the net flow accumulated in a separate 16-bit register. Through its 1-Wire interface, the DS2740 allows the host system read/write to access the status and current measurement registers. Each device has a unique factory-programmed 64-bit net address that allows it to be individually addressed by the host system. The interface can be operated with

standard or overdrive timing. Figure 2.4 illustrates the block diagram of the MAXIM DS2740 coulomb counter.

The DS2740 has two power modes: active and sleep. While in active mode, the DS2740 operates as a high-precision coulomb counter with current and accumulated current measurement blocks operating continuously and the resulting values updated in the measurement registers. Read and write access is allowed to all registers and the PIO pin is active. In sleep mode, the DS2740 operates in a low-power mode with no current measurement activity. Serial access to current, accumulated current, and status/control registers is allowed if $V_{DD} > 2V$. The DS2740 operating mode transitions from SLEEP to ACTIVE when:

1) $DQ > V_{IH}$, and $V_{DD} >$ UV threshold, or

2) V_{DD} rises from below UV threshold to above UV threshold.

The DS2740 operating mode transitions from ACTIVE to SLEEP when:

1) V_{DD} falls to UV threshold, or

2) SMOD = 1 and $DQ < V_{IL}$ for 2s.

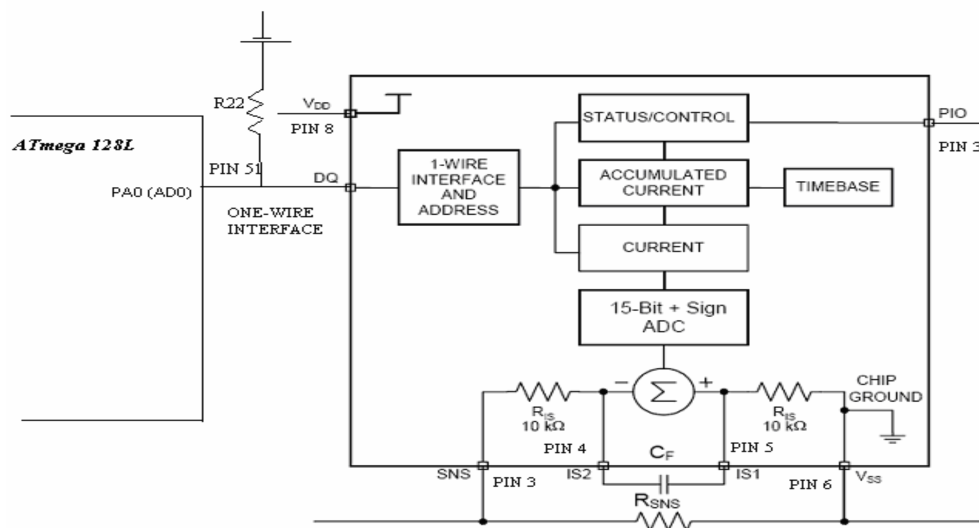


Figure 2.4 Block diagram of DS2740 [7, p3].

2.3.1 Current Measurement

In the active mode of operation, the DS2740 continually measures the current flow into and out of the device it is connected to by measuring the voltage drop across a low-value current-sense resistor, R_{SNS}. To extend the input range for pulse-type load currents, the voltage signal can be filtered by adding a capacitor between the IS1 and IS2 pins. The external capacitor and two internal resistors form a low-pass filter at the input of the ADC. The voltage-sense range at IS1 and IS2 is ±51.2mV. The input converts peak signal amplitudes up to 102mV as long as the continuous or average signal level (post filter) does not exceed ±51.2mV over the conversion cycle period. The ADC samples the input differentially at IS1 and IS2 with an 18.6 kHz sample clock and updates the current register at the completion of each conversion cycle.

The peripheral devices CC2420 and DS2740 are interfaced to the ATmega 128L using Serial Peripheral Interfacing and One-Wire interfaces respectively. The following chapter discusses the interfacing techniques in detail.

CHAPTER 3: INTERFACING

3.1 Introduction to Serial Peripheral Interfacing (SPI)

A Serial Peripheral Interface (also known as Four Wire interface [11]) can be used to interface memory, analog-digital converters, real-time clock calendars, processors and may other devices. SPI is a synchronous protocol in which all transmissions are referenced to a common clock generated by the master processor. The receiving peripheral uses the clock to synchronize its acquisition of the serial bit stream. Many chips may be connected to the same SPI interface of a master. A master selects a slave to receive by asserting the slave's chip select input. A peripheral that is not selected will not take part in the SPI transfer. All the peripheral devices are assigned a unique address. Figure 3.1 illustrate the interface between CC2420 and ATmega 128L.

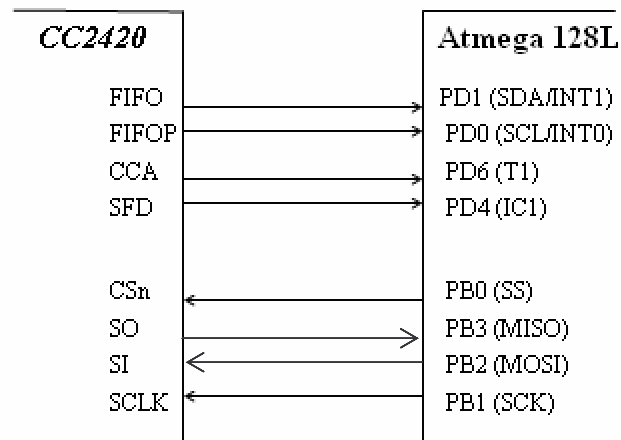


Figure 3.1 SPI interface between a CC2420 and an ATmega 128L

3.2 CC2420 Data Transfer Mechanism

The CC2420 is a slave device which is configured by the four SPI signals. The interface is used to read and wire buffered data. There are 33 16-bit configuration and status registers, 15 command strobe registers and two 8-bit registers to access the separate transmit and receive FIFOs. Each of the 50 registers is addressed by a 6-bit address. The RAM/Register bit (bit 7) must be cleared for register access. The Read/Write bit (bit 6) selects a read or a write operation and makes up the 8-bit address field together with the 6-bit address.

In each register read or write cycle, 24 bits are sent on the SI-line. The CSn pin (Chip Select, active low) must be kept low during this transfer. The bit to be sent first is the RAM/Register bit (set to 0 for register access), followed by the R/W bit (0 for write, 1 for read). The following 6 bits are the address-bits (A5:0). A5 is the most significant bit of the address and is sent first. The 16 data-bits are then transferred (D15:0), also MSB first.

The configuration registers can also be read by the microcontroller via the same configuration interface. The R/W bit must be set high to initiate the data read-back. CC2420 then returns the data from the addressed register on the 16 clock cycles following the register address. The SO pin is used as the data output and must be configured as an input by the microcontroller.

The timing for the programming is shown in Figure 3.2 with reference to Table 3.1. The register data will be retained during power down mode, but not when the power-supply is turned off (e.g. by disabling the voltage regulator using the VREG_EN pin). The registers can be programmed in any order.

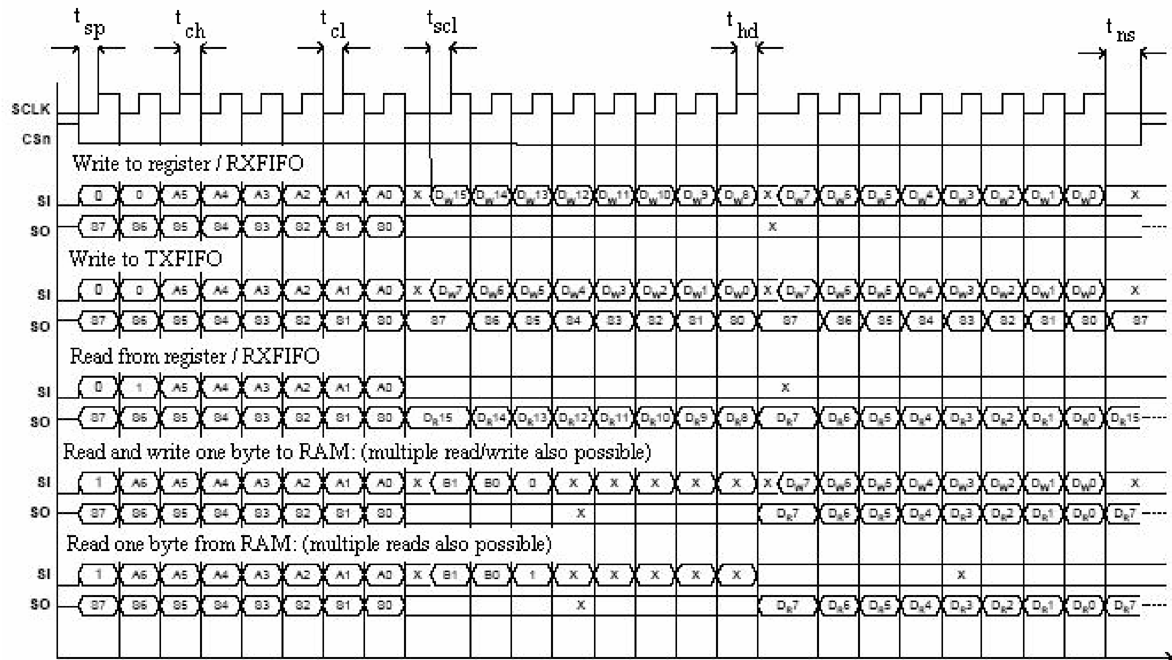


Figure 3.2 SPI Timing diagram [5, p26].

Parameter	Symbol	Min	Max	Units	Conditions
SCLK, clock frequency	F_{SCLK}		10	MHz	
SCLK low pulse duration	t_{cl}	25		ns	The minimum time SCLK must be low.
SCLK high pulse duration	t_{ch}	25		ns	The minimum time SCLK must be high.
CSn setup time	t_{sp}	25		ns	The minimum time CSn must be low before the first *positive* edge of SCLK.
CSn hold time	t_{ns}	25		ns	The minimum time CSn must be held low after the last *negative* edge of SCLK.
SI setup time	t_{sd}	25		ns	The minimum time data on SI must be ready before the *positive* edge of SCLK.
SI hold time	t_{hd}	25		ns	The minimum time data must be held at SI, after the *positive* edge of SCLK.
Rise time	t_{rise}		100	ns	The maximum rise time for SCLK and CSn
Fall time	t_{fall}		100	ns	The maximum fall time for SCLK and CSn

Table 3.1 SPI Timing specifications [5, p26].

The registers are accessed using various command strobes. The list of all the commands strobe signals are listed in CC2420 specifications.

3.2.1 CC2420 Interfacing Specifications

Figure 3.1 illustrates the connection between CC2420 and ATmega 128L. The microcontroller uses 4 I/O pins for the SPI configuration interface (SI, SO, SCLK and CSn). SO should be connected to an input at the microcontroller. SI, SCLK and CSn must be microcontroller outputs [6]. The microcontroller pins connected to SI, SO and SCLK can be shared with other SPI-interface devices. SO is a high impedance output as long as CSn is not activated (active low). CSn should have an external pull-up resistor or be set to a high level when the voltage regulator is turned off in order to prevent the input from floating. SI and SCLK should be set to a defined level to prevent the inputs from floating.

CC2420 operates in two modes, receive mode and transmit mode, depending upon the command strobes from the microcontroller.

3.2.2 Receive Mode

In receive mode, the SFD pin goes high after the start of frame delimiter (SFD) field has been completely received and goes low again after the last byte of the MPDU (MAC Protocol Data Unit) has been received [10]. If the received frame fails address recognition, the SFD pin goes low immediately. This is illustrated in Figure 3.3.

The data received or transmitted can be manipulated by accessing the RXFIFO and TXFIFO registers. The FIFO pin goes high when there are one or more data bytes in the RXFIFO. The first byte to be stored in the RXFIFO is the length field of the received frame, i.e. the FIFO pin is set high when the length field is written to the RXFIFO. The FIFO pin then remains high until the RXFIFO is empty. If a previously received frame is

completely or partially inside the RXFIFO, the FIFO pin will remain high until the RXFIFO is empty.

The FIFOP pin is high when the number of unread bytes in the RXFIFO exceeds the threshold. When address recognition is enabled the FIFOP pin will not go high until the incoming frame passes address recognition or if the number of bytes in the RXFIFO exceeds the programmed threshold. The FIFOP pin will also go high when the last byte

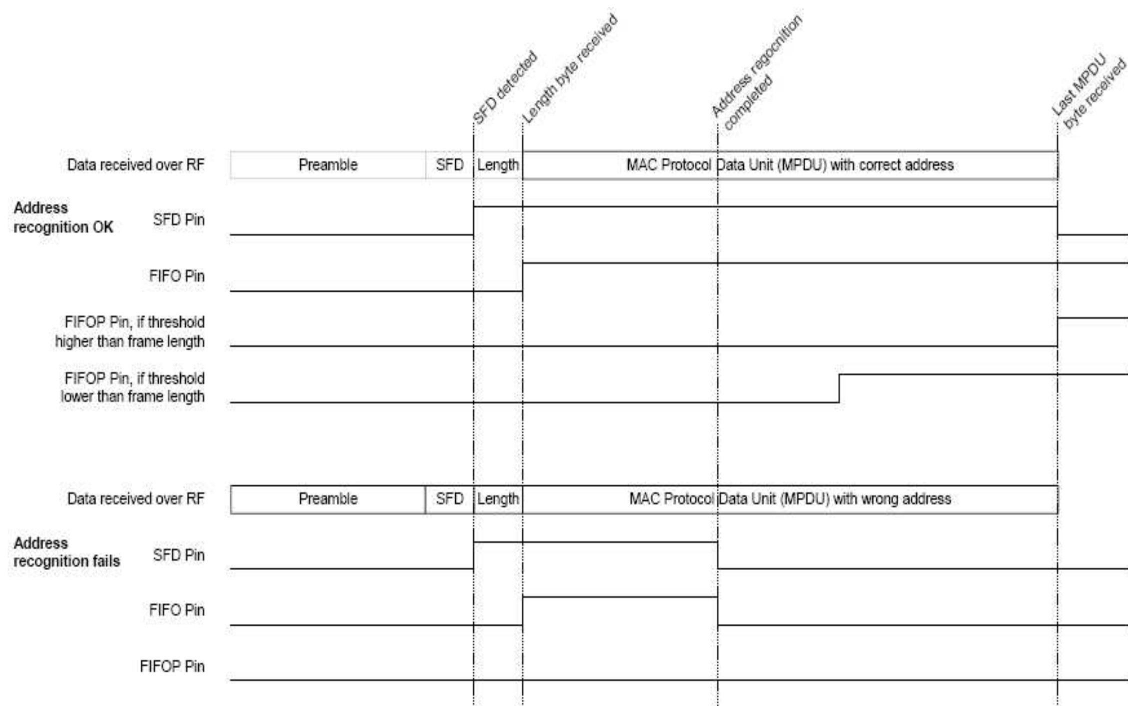


Figure 3.3 Pin activities during receive [6, p34].

of a new packet is received, even if the threshold is not exceeded. If so the FIFOP pin will go back to low once one byte has been read out of the RXFIFO.

When address recognition is enabled, data should not be read out of the RXFIFO before the address is completely received, since the frame may be automatically flushed by CC2420 if it fails address recognition. This may be handled by using the FIFOP pin, since this pin does not go high until the frame passes address recognition.

3.2.3 Transmit mode

During transmit the FIFO and FIFOP pins are still only related to the RXFIFO. The SFD pin is however active during transmission of a data frame, as shown in Figure 3.4. The SFD pin goes high when the SFD field has been completely transmitted. It goes low again when the complete MPDU (as defined by the length field) has been transmitted or if an underflow is detected.

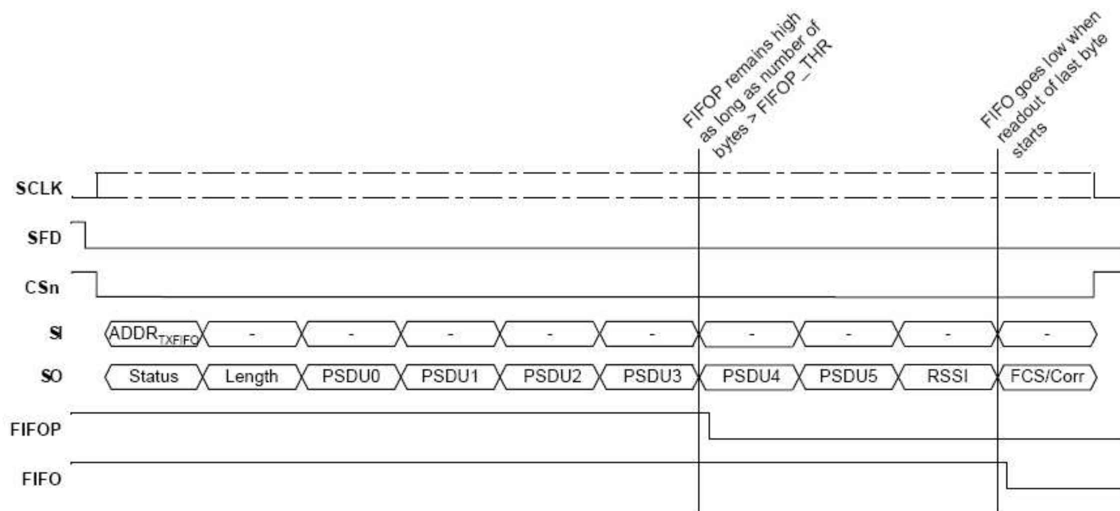


Figure 3.4 Example of pin activity when reading RXFIFO [6, p34].

As can be seen from comparing Figure 3.4 and Figure 3.5, the SFD pin behaves very similarly during reception and transmission of a data frame. If the SFD pins of the transmitter and the receiver are compared during the transmission of a data frame, a small delay of approximately 2 μs can be seen because of bandwidth limitations in both the transmitter and the receiver.

Figure 3.5 Pin activity example during transmit [6].

3.3 One-Wire Interface

The Dallas Semiconductor 1-Wire® bus is a simple signaling scheme that performs two-way communications between a single master and peripheral devices over a single connection. Figure 3.6 illustrates the pin diagram of DS2740 coulomb counter. The bus master is typically a microprocessor in the host system. Each device has a unique factory-programmed 64-bit net address that allows it to be individually addressed by the host system, supporting multiple connections to 1-Wire bus. The interface can be operated with standard or overdrive timing [8].

Figure 3.6 Pin Configuration of DS2740

OVD 1-Wire Bus Speed Select
PIO Programmable I/O Pin
SNS Sense Resistor Input
IS2 Current-Sense Input
IS1 Current-Sense Input
Vss Current-Sense Resistor Return
DQ Data Input/Output
V_{DD} Power-Supply Input (2.7V to 5.5V)

Dallas Semiconductor's 1-Wire communication protocol can easily be implemented on almost any microcontroller:

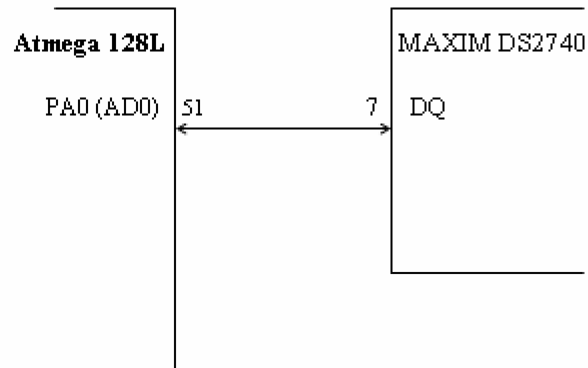


Figure 3.7 One-Wire Interface between ATmega128L and MAXIM DS2740

- Only two bidirectional PIO states are necessary: high impedance and logic low. If a bidirectional pin is not available on the bus master, separate output and input pins can be connected together.
- The 1-Wire timing protocol has specific timing constraints that must be followed in order to achieve successful communication. The DS2740 can operate in two communication speed modes, standard and overdrive. The speed mode is determined by the input logic level of the OVD pin.
- The 1-Wire bus must have a pull-up resistor at the bus-master end of the bus. For short line lengths, the value of this resistor should be approximately 5kΩ. The idle state for the 1-Wire bus is high.

The protocol for accessing the DS2740 is as follows 1) Initialization - Reset. 2) Net Address Command. 3) Function Command followed by Transaction/Data.

1. The start of any 1-Wire transaction begins with a reset pulse from the master device followed by a simultaneous presence detect pulses from the slave devices.
2. Once the bus master has detected the presence of one or more slaves, it can issue one of the Net Address Commands.

Search Net Address [F0h]. This command allows the bus master to use a process of searching to identify the 1-Wire net addresses of all slave devices on the bus. In multi-drop systems this command must be used first, then Match Net Address.

Match Net Address [55h]. This command allows the bus master to specifically address one DS2740 on the 1-Wire bus. Only the addressed DS2740 responds to any subsequent function command.

Read Net Address [33h or 39h]. This command allows the bus master to read the DS2740's 1-Wire net address. This command can only be used if there is a single slave with correspondent opcode on the bus. Bit **RNAOP**, responsible for that opcode must be set first in a system of two DS2740s.

Skip Net Address [CCh]. This command saves time when there is only one DS2740 on the bus by allowing the bus master to issue a function command without specifying the address of the slave.

Resume [A5h]. This command increases data throughput in multi-drop environments where the DS2740 needs to be accessed several times. After successfully executing a Match Net Address command or Search Net Address command, an internal flag is set in the DS2740. When the flag is set, the DS2740 can be repeatedly accessed through the Resume command function. Accessing another device on the bus clears the flag, thus preventing two or more devices from simultaneously responding to the Resume command function.

3. After successfully completing one of the net address commands, the bus master can access the features of the DS2740 with any of the Function Commands.

Read Data [69h, XX]. This command reads data from the DS2740 starting at memory address XX. The LSb of the data in address XX is available to be read immediately after the MSb of the address has been entered.

Write Data [6Ch, XX]. This command writes data to the DS2740 starting at memory address XX. The LSb of the data to be stored at address XX can be written immediately after the MSb of address has been entered. Incomplete bytes are not written.

CHAPTER 4: TESTING METHODOLOGY

4.1 Introduction

Figure 4.1 illustrates a simple flowchart of a hardware life cycle. Every stage in product development cycle is verified, validated and reviewed to reduce the errors caused during production. Some defects/faults escape the rigorous inspection. This may be due to design errors, manufacturing defects or environmental factors. Almost 30-50% of fabrication costs in electronics production is caused by testing and repair operations [13].

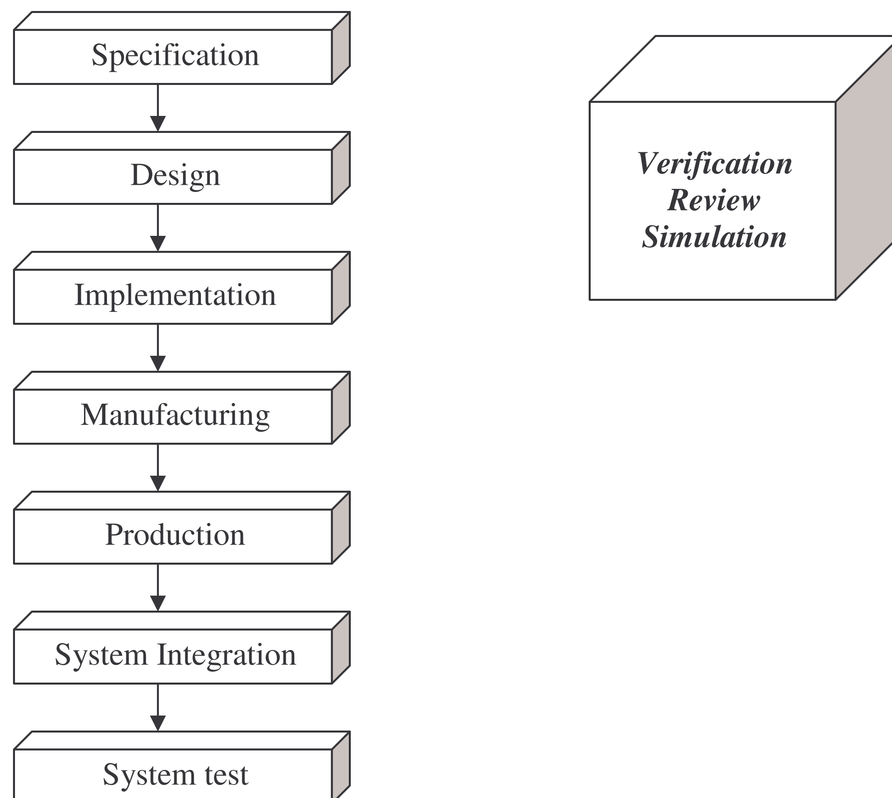


Figure 4.1 Hardware life cycle

This section gives an introduction to the types of faults found in the PCB and the main categories of test methods. It is followed by a detailed description of the test agenda and the steps carried out to test the wireless evaluation board under test.

PCB flaw detection can be classified broadly into two categories; electrical/contact methods and non-electrical/non-contact methods. In the industry many types of tests are implemented depending on design and cost constraints. Some of the common methods include bed of nails, functional test, in-circuit test, X-ray detection, 3-D laser detection and so on. The high complexity of hardware structure with limited test port presents a major challenge in testing an embedded system. Functional and in-circuit test enables testing system of reasonable complexity with considerable accuracy.

In a functional test, the board is powered and an output signal is characterized by application of specific input stimuli [22]. The output signal is measured and compared against the expected results. This test is employed to validate the functional specifications of the board. Though it yields very high fault coverage, the major drawback of not identifying the faults at component level remains.

In-circuit testing is carried out by applying stimuli and measuring the signal nodes on the circuit board without powering the entire board.

The PCB under test was designed effectively for testability. Appropriate test headers are added to observe the input/output signals on different ports. JTAG support by ATmega 128L eased the difficulty of verifying the signals on user input/output ports. UART interfaces, LEDs and push buttons further enhanced the functional testability of the board.

As discussed in earlier chapters, the essential modules are the ATmega 128L, CC2420 and DS2740. The functionalities of these modules are tested by observing the output upon downloading the appropriate executables onto the RAM. This method is carried out extensively to test the individual functionality of the modules as well as the overall functionality of the boards after integration of the modules. All the major signals are also analyzed to determine the accuracy of the board.

A detailed test plan was formulated after repeated design reviews. The architecture of the board was analyzed and a test agenda was created. The power supply, external crystal and the reset signal constitutes the main infrastructure of the board. Thus, the initial test phase is carried out to ensure proper functioning of the above described test areas. The next logical step was to test the JTAG interface. It represents the gateway to download the compiled machine code from a PC onto the ATmega 128L. Observing user I/O signals followed by UART interface tests covers one-third of the test plan. The final phase of testing includes validating the CC240 and DS2740. Every test is explained in a detailed manner in the following sections.

The order, in which the various features were tested, can be summarized as follows:

i. Power circuit
ii. Crystal oscillator
iii. Reset signal
iv. JTAG interface
v. User I/O signal
vi. UART Interface
vii. Test for CC2420

viii. Test for DS2740

4.2 Power Circuit

ATmega 128L, CC2420 and DS2740 are supplied with a 3.3V supply. The Figure 4.2 gives the layout of the un-routed embedded evaluation board. Micro-controller, RF and Coulomb counter modules can be identified. Measuring the supply voltage at specific modules does not guarantee the accuracy of the power circuit. Fabrication errors may be a cause for high fluctuations at different operating conditions. To ensure proper operation of the power circuit, the voltage across the all the modules are tested for an extended period of time at various stages of operations.

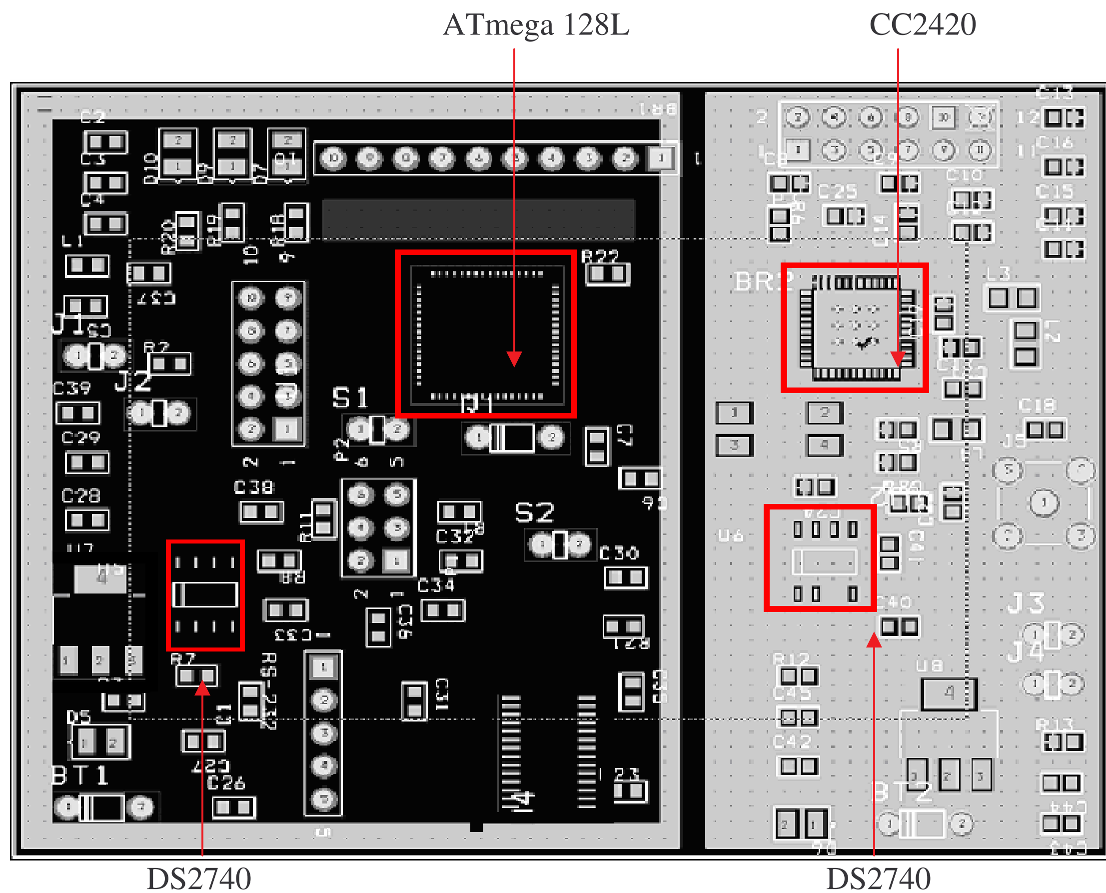


Figure 4.2 Location of the ATmega 128L, CC2420 and DS2740 modules

4.2.1 Testing ATmega 128L Power Supply

According the ATmega 128L specifications [1], a low pass filter is connected between AVCC PIN64 and VCC PIN21, PIN52 for proper functioning of the microcontroller module. The voltage supplied to ATmega 128L was measured during the start, transmit and receive operations. The voltage across the Vcc PIN 21 and GND PIN 22 was measured and the results confirmed proper voltage readings of 5.5 V. Figure 4.3 illustrate the location of the power circuit for ATmega 128L.

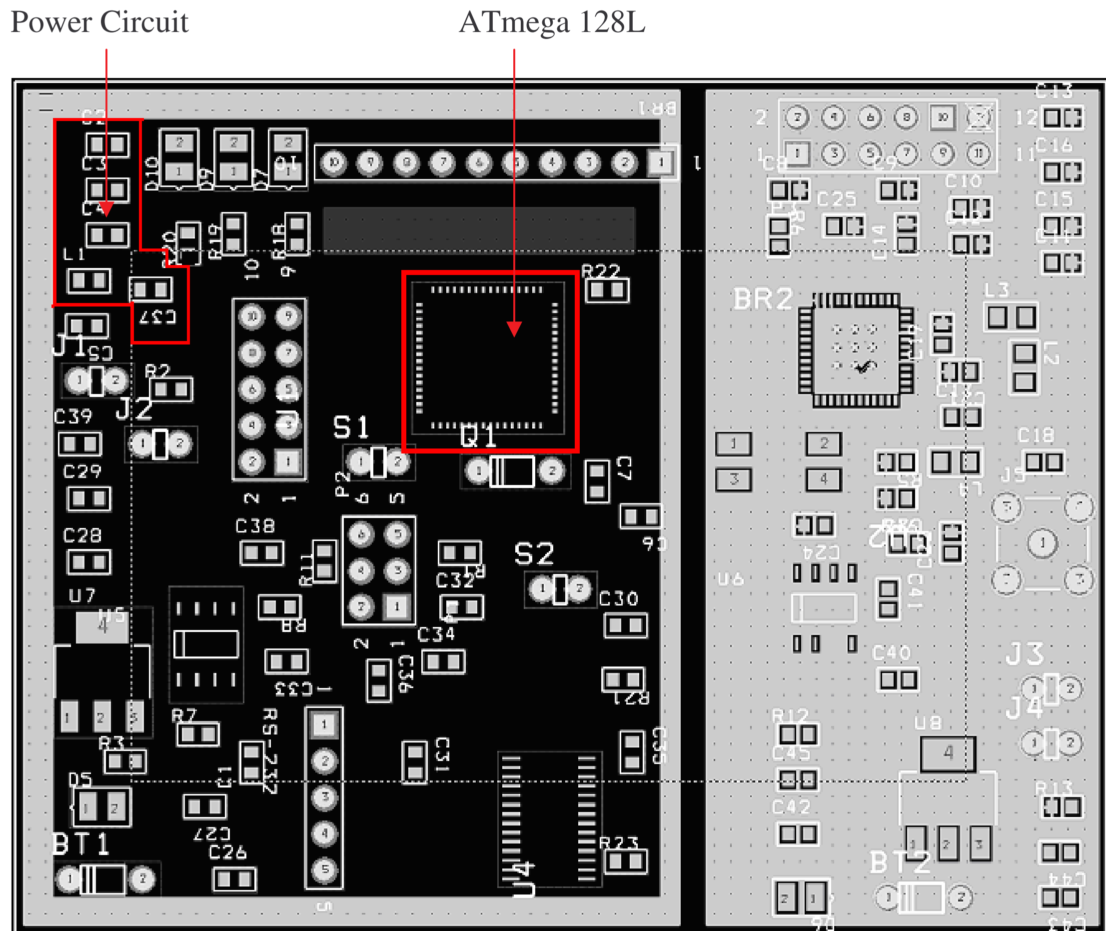


Figure 4.3 Location of Power circuit for ATmega 128L

4.2.2 Testing CC2420 Power Supply

Proper power supply must be provided for error free performance of the CHIPCON transceiver. The power circuit for the RF module is designed based on the reference design provided by the CHIPCON specifications [2]. The circuit was closely followed to obtain optimum performance of the RF module. Figure 4.4 shows the reference design by CHIPCON support. In order to test the power circuit, the voltage across VREG_IN PIN 43 and GND PIN 9 was measured and the results confirmed proper voltage readings of 3.5-3.9 V. Figure 4.5 illustrate the location of the power circuit for the RF module.

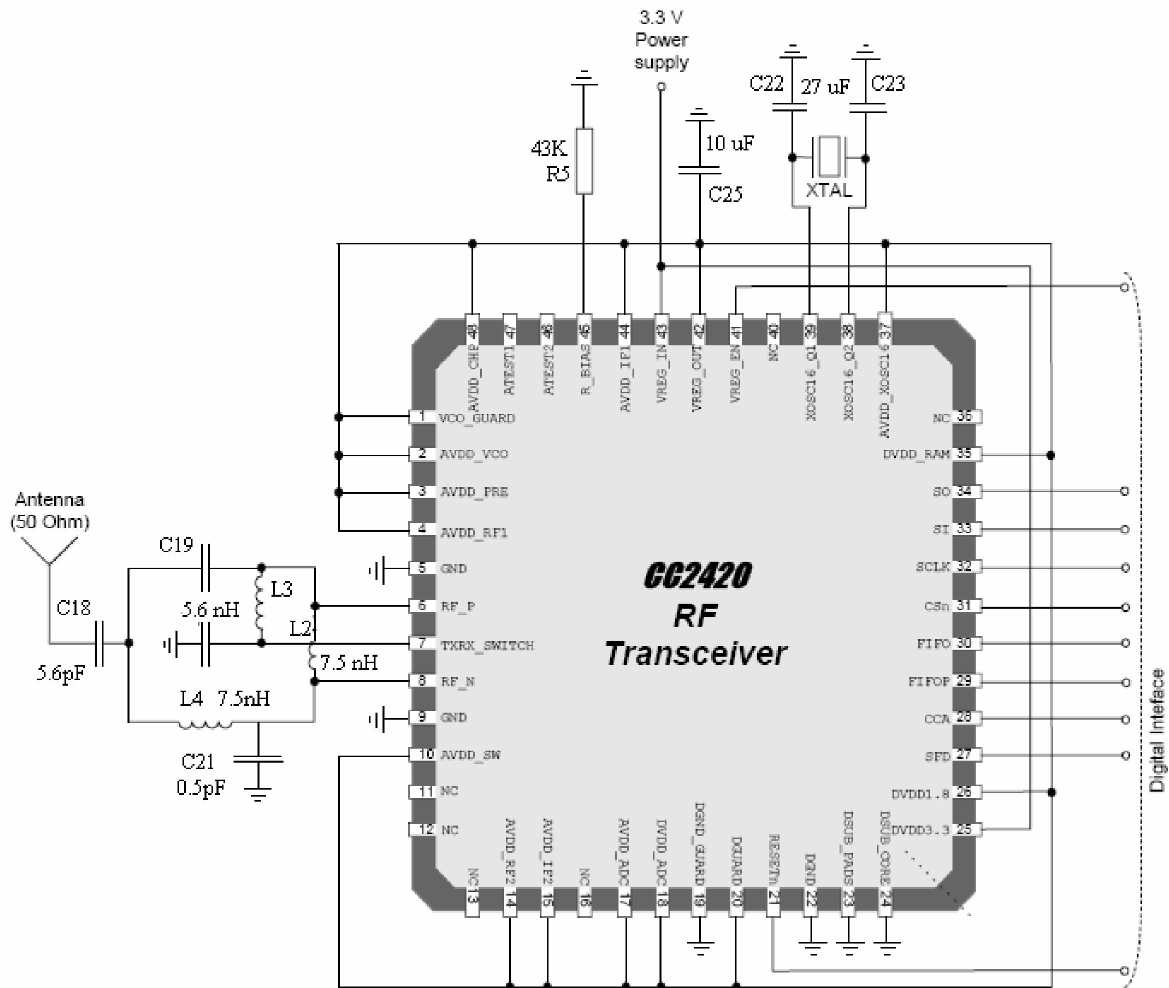


Figure 4.4 CC2420 reference design provided by CHIPCON support [6]

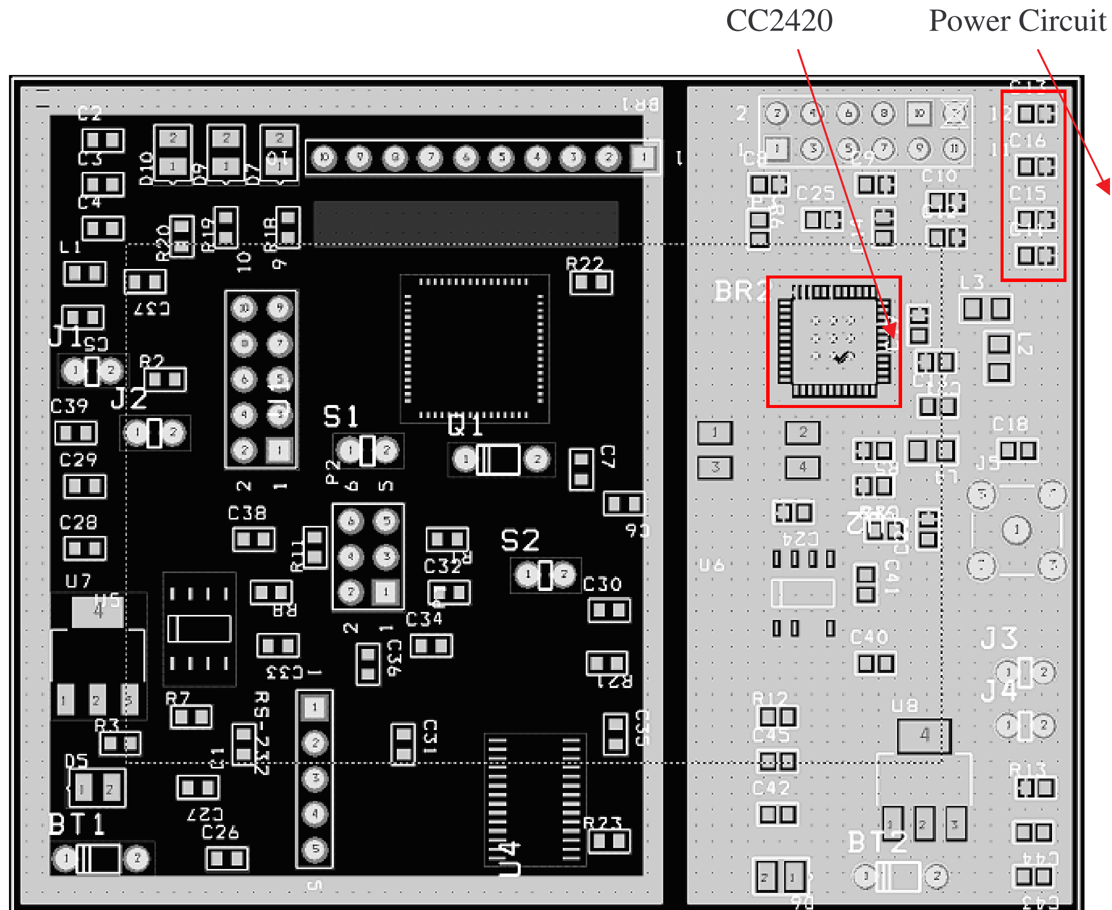


Figure 4.5 Location of Power circuit for CC2420

4.2.3 Testing DS2740 power supply

DS2740 is supplied with a 3.3V power supply at Vdd through a 150Ω resistor. MAXIM provided a reference design which was followed to design the power circuit. Figure 4.6 depicts the reference design provided by MAXIM support. The voltage across Vdd PIN 8 and Vss PIN 6 was measured and the results confirmed proper voltage readings of 3.3-3.5 V. The location of the power circuit can be seen on Figure 4.7.

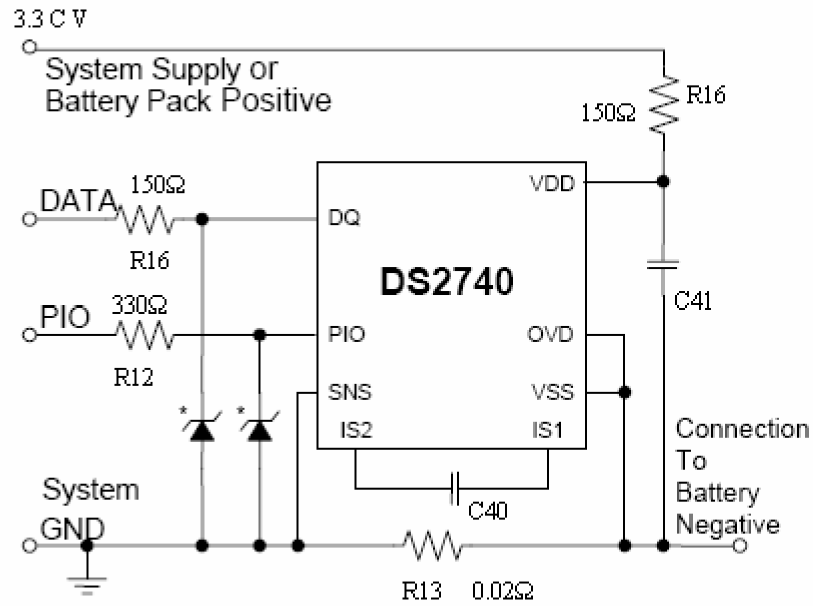


Figure 4.6 Reference design for Coulomb counter DS2740 provided by MAXIM [8].

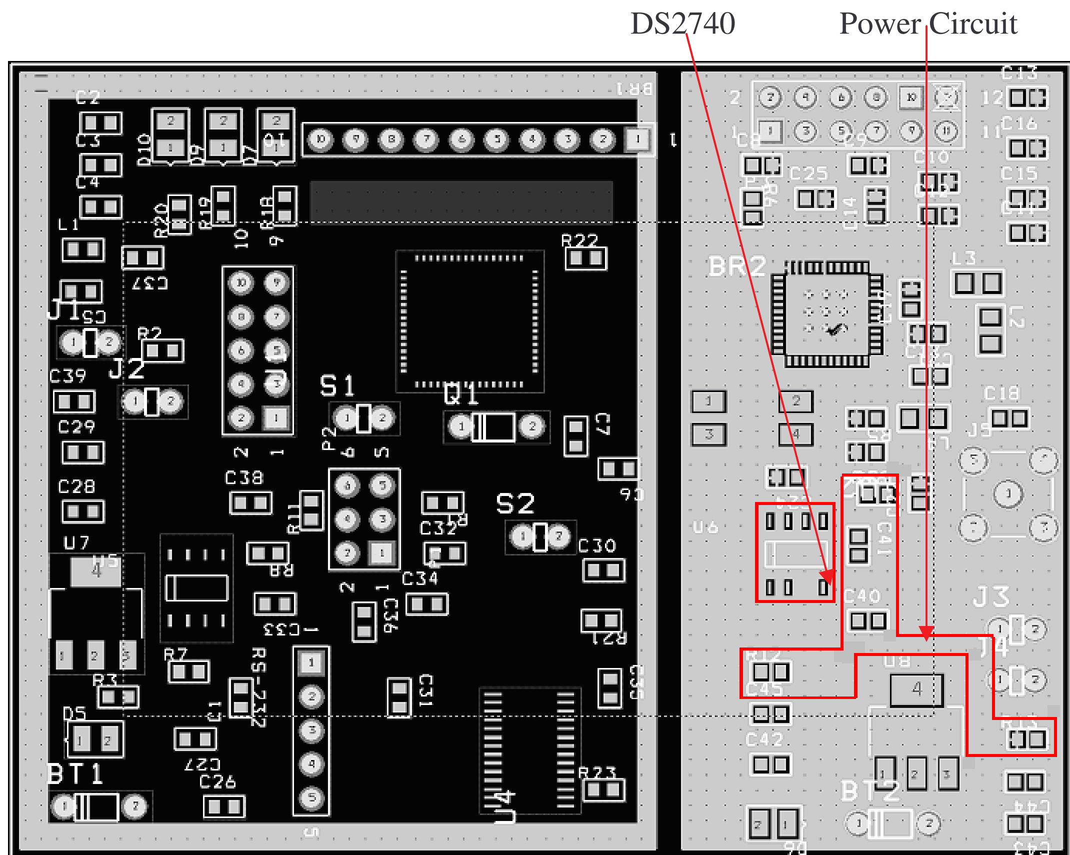


Figure 4.7 Location of Power circuit for DS2740

4.3 Crystal Oscillator

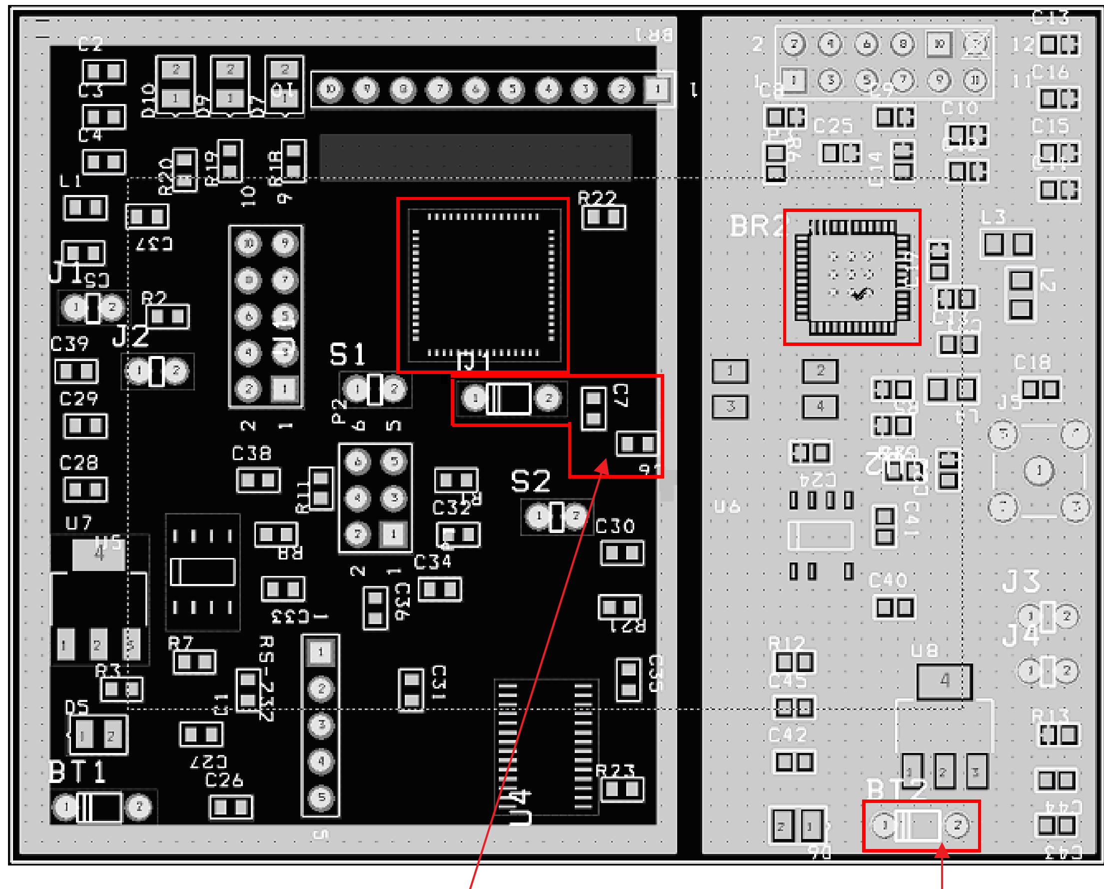
The embedded board consists of two external crystal oscillators connected to ATmega 128L and CC2420. Since the functioning of the microcontroller and the transceiver is based on accurate functioning of the crystal oscillator is it necessary to ensure the accuracy of the oscillators on board. The following sub sections gives details of the tests carried out to verify the crystal oscillators. Figure illustrates the location of the crystal oscillators connected to ATmega 128L and CC2420.

4.3.1 Test for Crystal Oscillator connected to ATmega 128L

An external crystal oscillator is connected between XTAL1 PIN 24 and XTAL2 PIN 23 of ATmega 128L. It can operate at four different frequencies viz. 1MHz, 4MHz, 8MHz and 16MHz depending on the fuse settings of the microcontroller. Since the whole operation of the embedded wireless board is configured at 8MHz, the crystal oscillator was set to operate at 8 MHz and the clock pulses were observed using an oscilloscope. A stable clock pulse was observed in the oscilloscope. Figure 4.8 gives the location of the crystal oscillator.

4.3.2 Test for Crystal Oscillator connected to CC2420

A 16MHz crystal oscillator is connected between XOSC_Q1 PIN 39 and XOSC_Q2 PIN 38 of CC2420. The embedded board was powered and the clock pulses were observed to be stable using an oscilloscope. The location of the crystal oscillator is show in Figure 4.8.



Crystal Oscillator for ATmega 128L Crystal Oscillator for RF module

Figure 4.8 Location of external crystal oscillator for ATmega 128L and CC2420

4.4 Reset Circuit

The reset circuit only consists of a push button S1 and few passive components. The reset signal can be generated by two mechanisms: by closing the circuit using the push button and by reducing the supply voltage to fall below the threshold value. In order to test the reset circuit these mechanisms were tested.

The IRESET PIN 20 was observed while the push button is pressed. The supply voltage was reduced below 5V and the reset signals were observed.

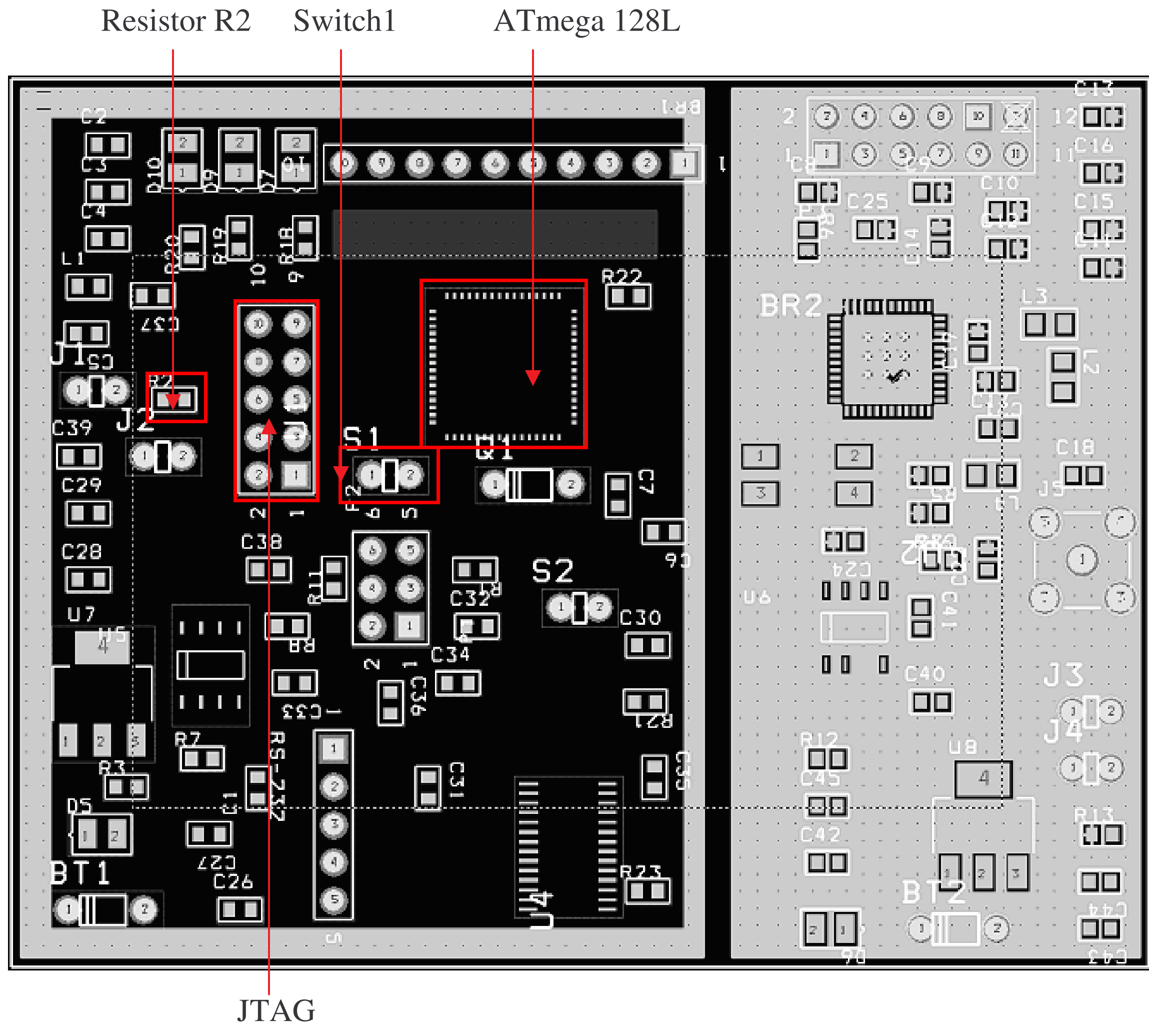


Figure 4.9 Location of JTAG port and Switch1

4.5 JTAG Interface

The JTAG circuit in the PCB is compatible with the JTAG ICE MK II [2] connector. Figure 4.9 indicates the location of the JTAG circuit. The main purpose of this test is to authenticate the JTAG signals on board. AVR Studio 4.0 was used as a programmer throughout the test phase. The Auto Detect function of the programmer was used to probe the JTAG chain. When the board was powered and the JTAG connected, the device was successfully detected by AVR Studio 4.0.

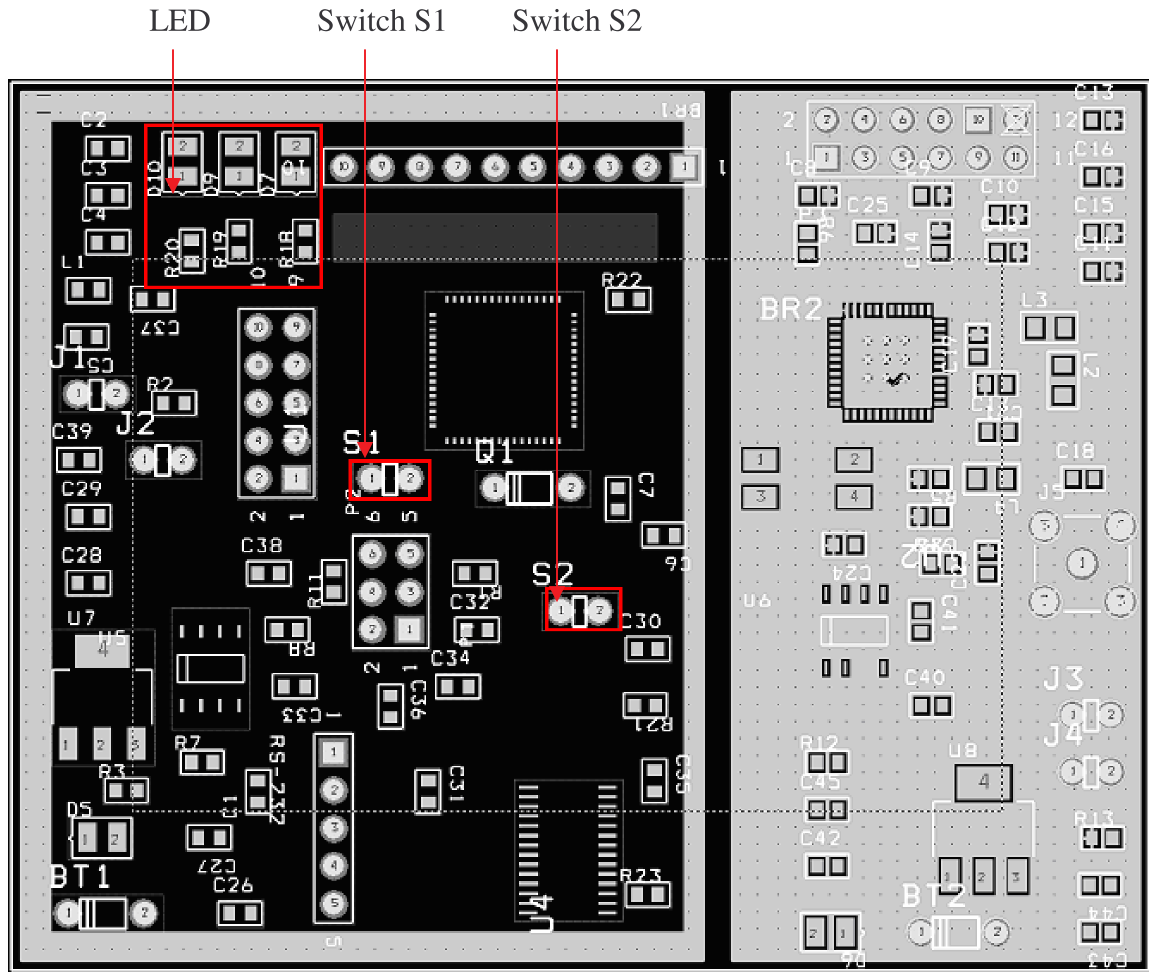


Figure 4.10 Location of LEDs and switches

4.6 User I/O

User I/O section consists push buttons, LEDs and 3 headers to observe the signals in different Port pins. In order to test these I/O signals various simple codes were written, compiled and downloaded to test the functionality of these signals. Figure 4.10 give the location of the switch S1 and S2

4.6.1 Test for Push Button

- Test code Switch.c was written to test the working of the push button. The code upon downloading makes the push button activate PORT PG0, PG1, PG2

associated to the LEDs. The LEDs operated as expected depending on the switch press.

- Ø Another test code module Switch_Interrupt.c was written make the Switch act as an external interrupt to PORT PE5. The LEDs were made to switch ON and OFF depending on the external interrupt.

4.6.2 Test for LED

The test written for the push button also tests the connections for LEDs. However, test code was written to set the timer interrupt for various time intervals and the interrupt was observed with the help of LED status. Figure 4.10 gives the location of LED D7, D9, D10.

- Ø A LED_Timer_Final.c was written to test the timer functionality of ATmega 128L. The timer was set to occur every one second which eventually activated the LEDs. One-second blinking was observed.

4.6.3 Test for MSV5

MSV5 is a 5 pin header connected to MAXIM 3243. To test the connections between the Port Pins associated with the microcontroller and the header, each pin has to be activated. MAXIM 3243 amplifies the signals making it compatible with the RS 232 [13]. The pin header represents RS232 signals; RTS, RD, TD, CTS. Test code MSV5.c was written and downloaded to activate the respective signals associated with PORT PIN PD5, PD2, PD3 and PD7. The voltage level indicated proper connection.

4.6.4 Test for MSV5X2

MSV 5X2 header is a 10 pin header connected to Port F and RS 232 control signals. MSV5X2.c was written to test the reliability of the connections to this header. Upon

downloading the code onto the microcontroller, the port pins PF0, PF1, PF2, PF3, TXD1, RXD1, CTS and RTS were made HIGH. The voltage across each pin was measured to verify the connections.

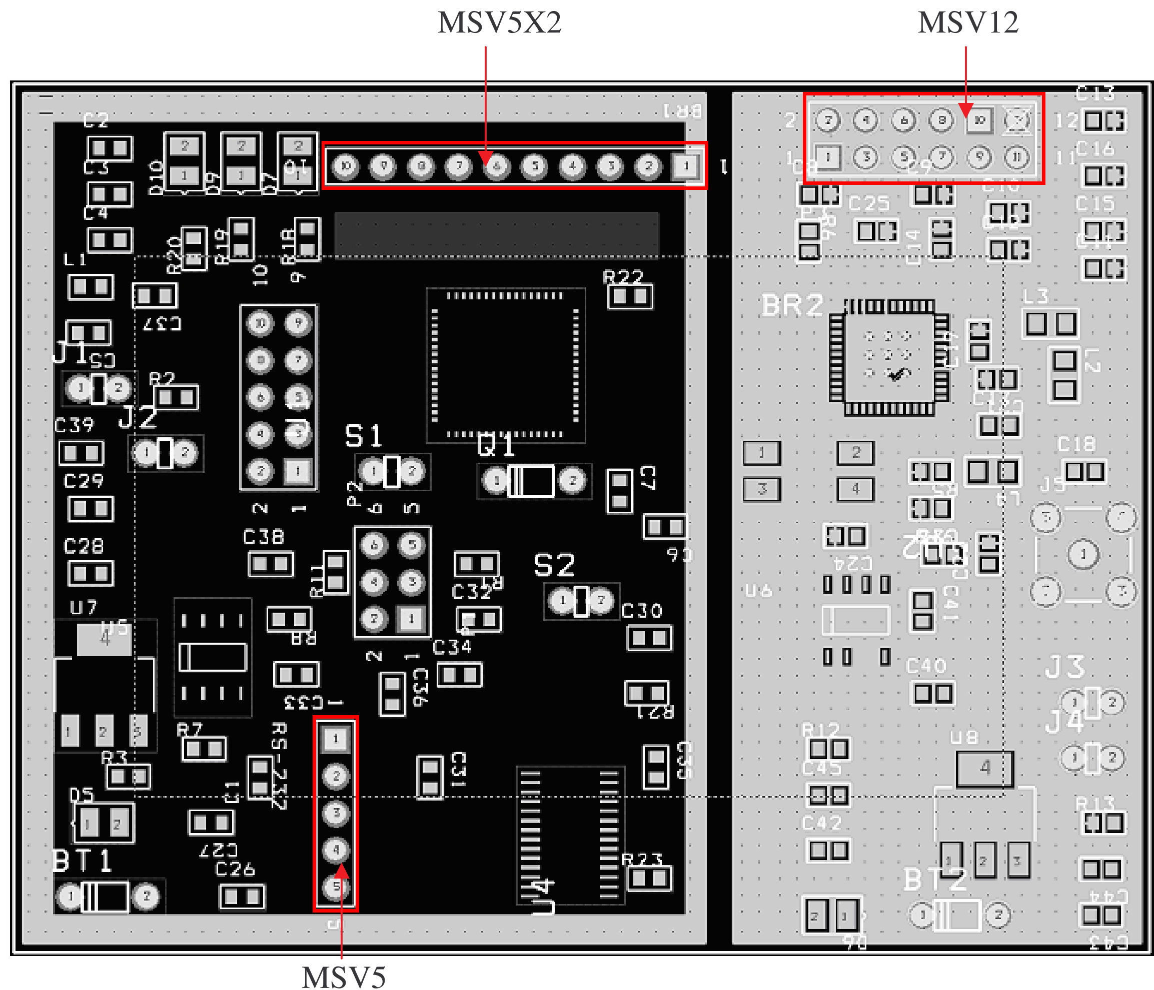


Figure 4.11 Location of headers MSV5, MSV5X2 and MSV12

4.6.5 Test for MSV12

MSV12 header is a 12 pin header; the main purpose of this header is to evaluate the SPI interface. All the signals associated with the SPI interface between ATmega 128L and CC2420 can be observed at this header. There are two ways by which we can test the signal. First method involves downloading the final firmware onto the board and

observing the timing signals of SPI interface. Second method involves downloading MSV12.c to activate each PORT PIN associated with SPI to evaluate the connection. Both methods were carried out to ensure proper connection status.

4.7 UART Interface

UART Interface plays an essential role in evaluating the operation of the board. It acts as a window to evaluate the working of the board. UART_TEST.c was written and tested to verify the interface. The code takes input from the keyboard and displays the key press on the hyper terminal at different baud rates. This simple code was downloaded and errorless operation was observed at different baud rates.

4.8 Test for CC2420

The CC2420 RF transceiver module drives the RF communication. The test code was first tested on the CC2420DBK board to ensure that the code works as per the specified functionality. The input/output port variables were changed according to the requirements of the board under test and were downloaded. A burst of characters were sent and received by a similar board configured as a receiver. The received characters were displayed on a PC running Hyper-Terminal at a fixed baud rate. At the receiver the RXFIFO register buffers all the character and displays them at the specified baud rate. Figure 4.12 depicts the set up for the CC2420 test.

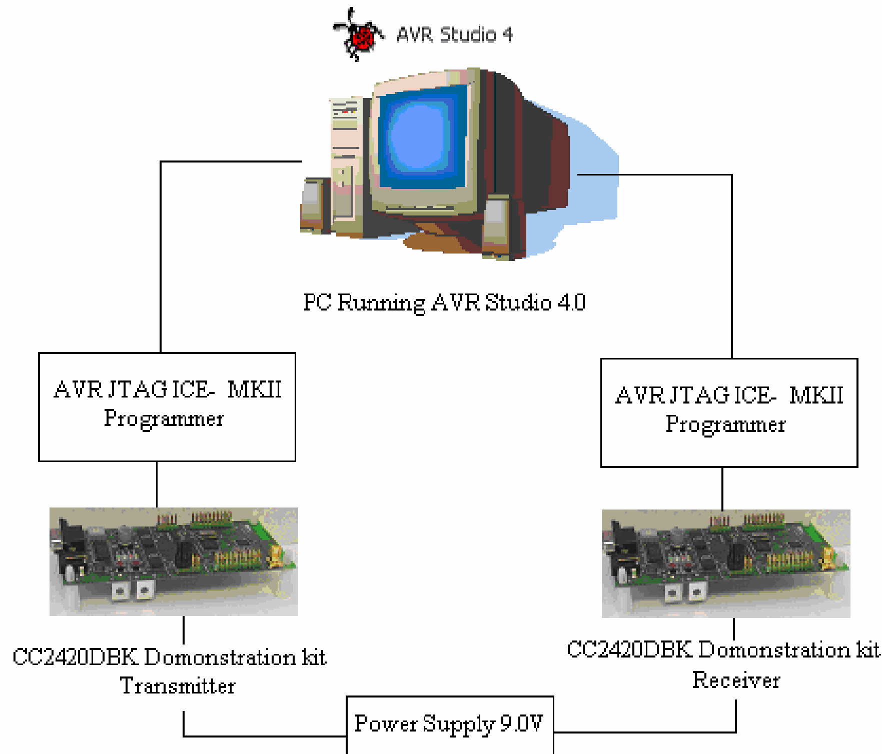


Figure 4.12 Block diagram of the experimental setup for CC2420 test

4.9 Test for DS2740

DS2740 module measures the current drawn by the microcontroller during transmission. In order to test the circuit operation, DS2740.c was downloaded and tested. Upon downloading the code, the microcontroller ATmega 128L polls for the coulomb counter DS2740. Once the device is recognized, the status is displayed on the hyper-terminal. This code was used to test the one-wire interface compatibility of the slave device. The current measured by the coulomb counter was tested against standard measurement techniques. Figure 4.13 gives a diagrammatic representation of the test setup.

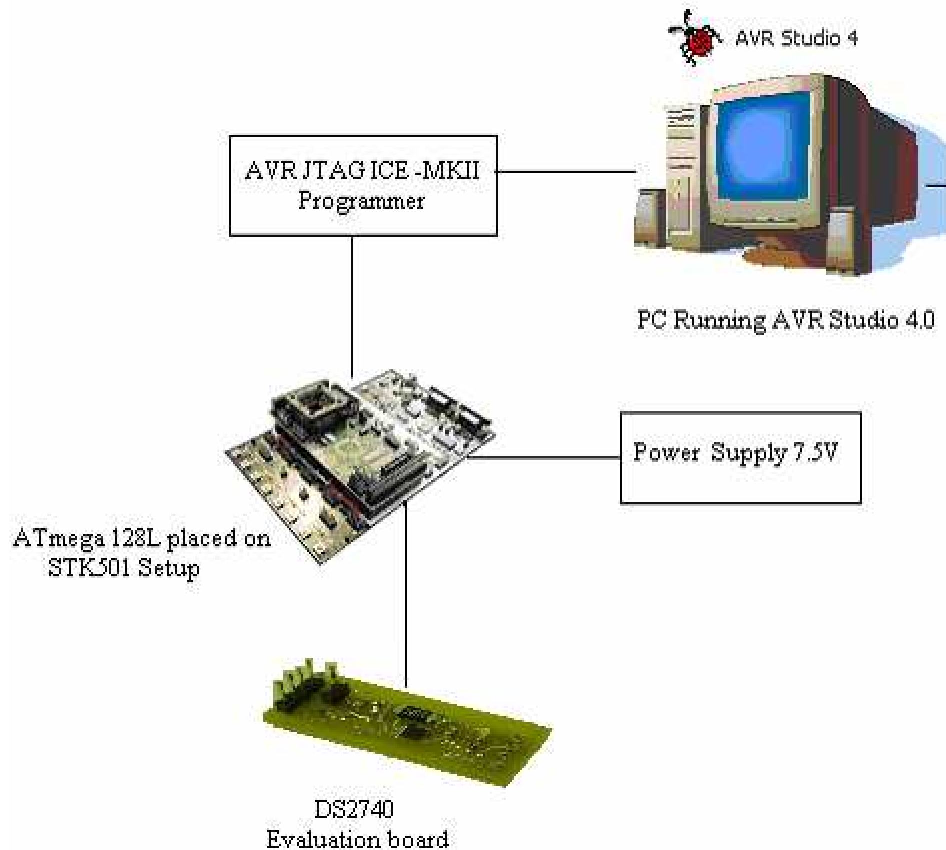


Figure 4.13 Block diagram of the experimental setup for DS2740 test

4.10 Testing full functionality of the board

Figure 4.14 presents a diagrammatic representation of the set up to test the full functionality of the board. A prototype was developed to test the Full-Functionality_test.c. The set up consists of a CC2420DBK board, which consists of CC2420 and ATmega 128L, and a DS2740 Coulomb counter evaluation board. The CC2420DBK board is powered by a 7.5V regulated supply. The Full-Functionality_test.c code is downloaded onto the ATmega 128L on the board. Upon successful completion of the initial process, the slave device, DS2740 evaluation board is recognized by the ATmega 128L microcontroller and the status is updated on the Hyperterminal. One of the

boards is configured as a transmitter, while the other as a receiver. A switch S1 is pressed on CC2420DBK board to transmit the content of the current accumulator register of DS2740 sent. The CHIPCON CC2420 RF transceiver drives the RF communication. During transmission, the Start of Framer Delimiter (SFD) pin remains high until the transmission is over. And then goes low on completion of transmission. The receiver also behaves in a similar fashion. The code is programmed in way that the SFD pin acts as a flag for measuring the current from the DS2740 evaluation board. As soon as the SFD pin goes logic high, the ATmega 128L reads the content of the current accumulator register and transmits the register vale. At the receiver end, the value receiver is displayed continuously on the hyper terminal.

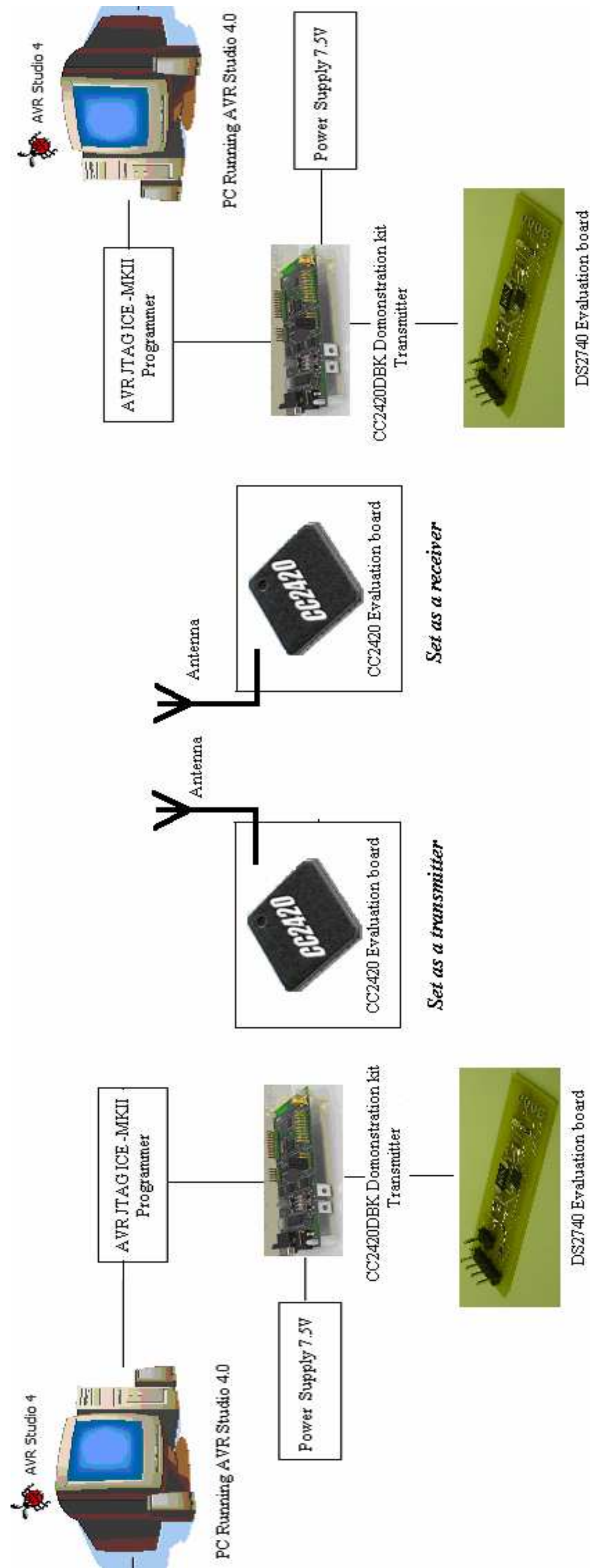


Figure 4.14 Set up to test full functionality of the board.

CHAPTER 5: CONCLUSION

The main objective of the thesis is to test a newly designed wireless embedded evaluation board compliant with 802.15.4. The design schematic was imitated on a prototype and verified by implementing several test cases. The flaws identified during the test cycle were eliminated by redesigning the appropriate location of the faulty circuit.

The simple approach involved testing the individual modules was undertaken. The test plan was developed and systematically followed throughout the test cycle. The test cycle focuses on the micro controller module, RF module and the coulomb counter modules in particular. The test methodology was structured in order to identify any possible faults that may occur in the board.

The initial test phase involved testing the ATmega 128L microcontroller module, CC2420 RF module and DS2740 coulomb counter module as individual functional blocks. Several test cases were developed to test the modules. The code written was first tested on a prototype to ensure that the code was errorless. The micro-controller module was emulated with the help of STK500, STK501 and JTAG ICE MKII connecter setup illustrated in the Figure 5.1. Figure 5.2 illustrates CHIPCON CC2420DBK and Figure 5.3 illustrates CC2420EB boards that were used as prototype to test the test cases for RF module. DS2740 evaluation board is illustrated in the Figure 5.4 was used to verify the test cases for DS2740 module. The board consists of header, user I/O interface like Switches and LED. Simple test codes were written to check the connection between the

headers, LEDs and switches to the respective pins on the ATmega 128L or CC2420 or DS2740.

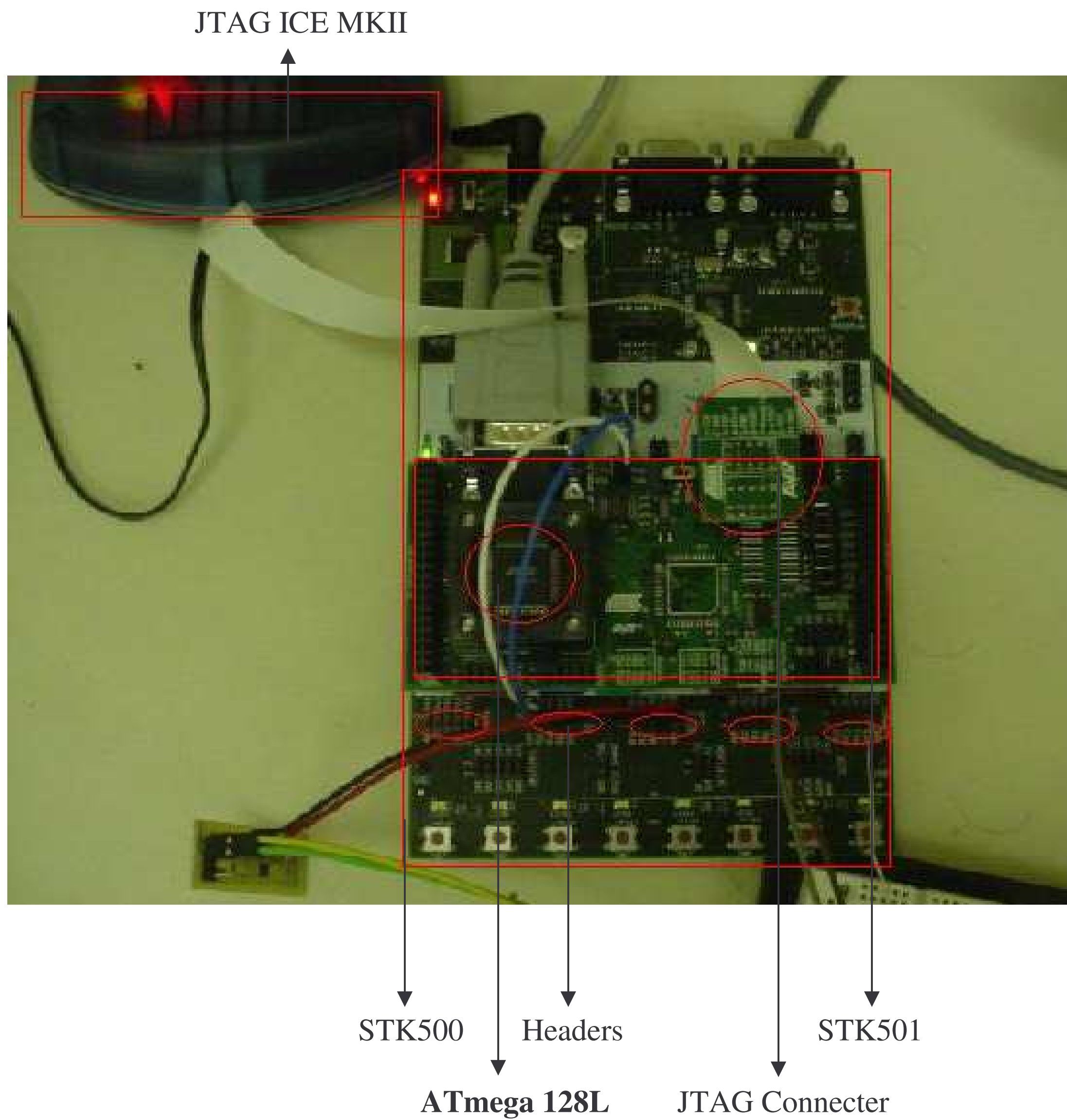


Figure 5.1 ATmega 128L microcontroller module setup.

RS232 Connecter to display the characters on the Hyper-terminal

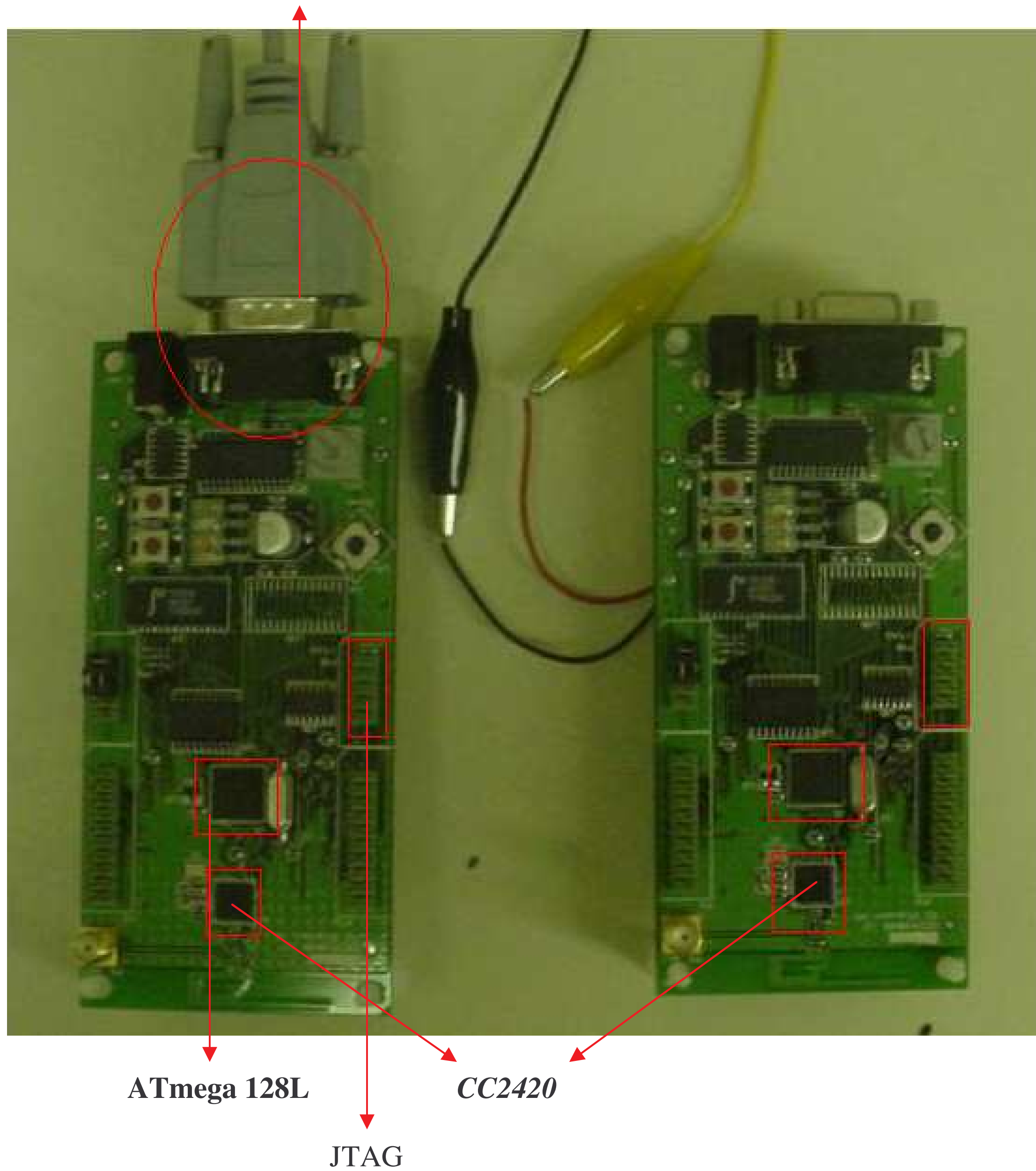


Figure 5.2 CC2420 module set up using CHIPCON CC2420DBK evaluation board.

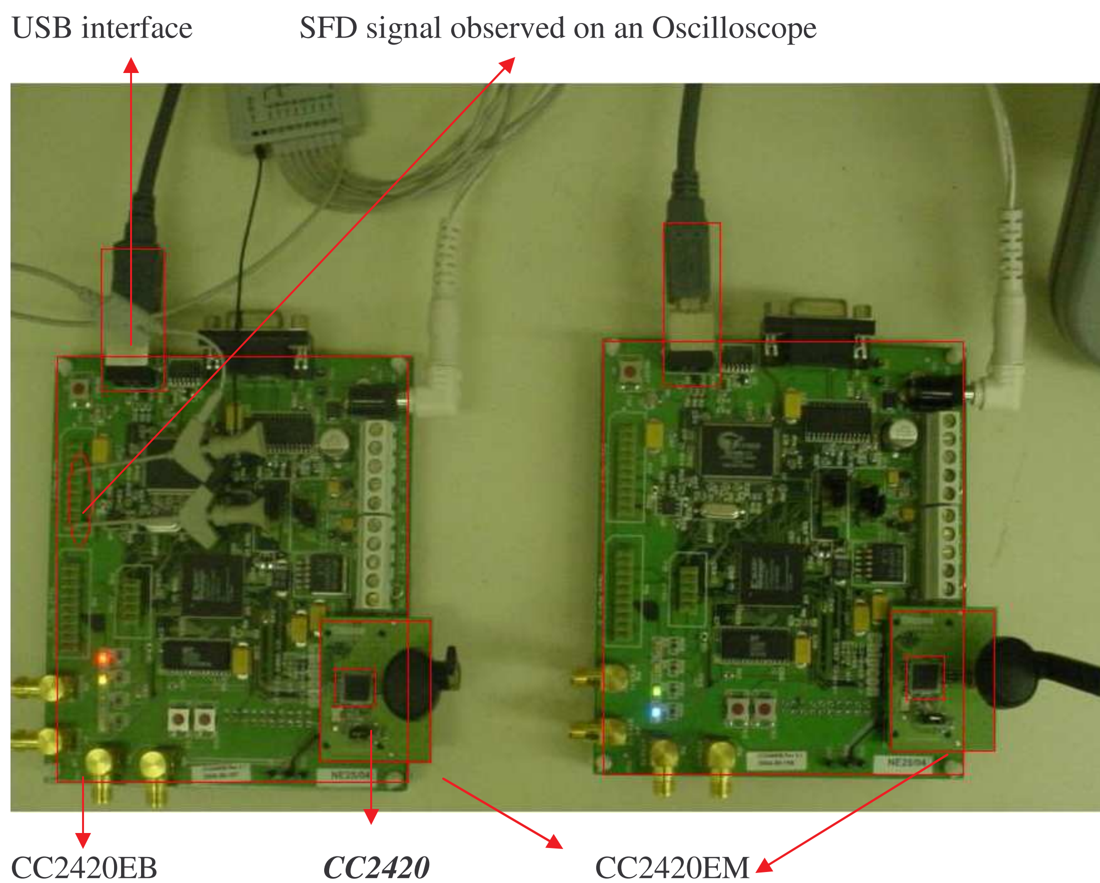


Figure 5.3 CC2420 module set up using CHIPCON CC2420EB evaluation board.

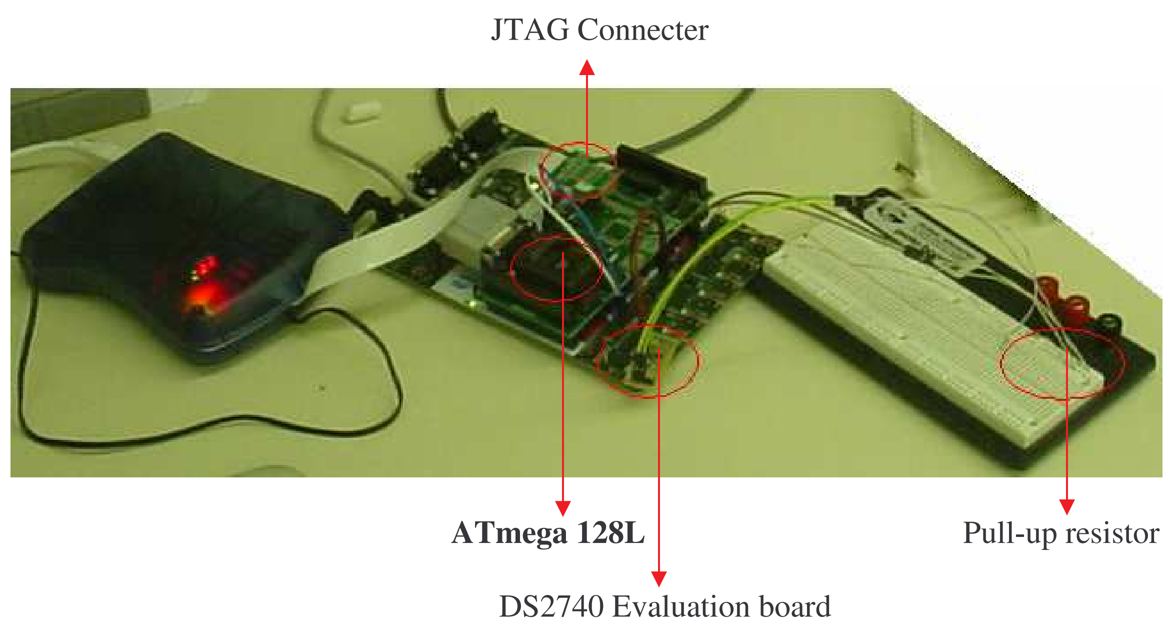


Figure 5.4 DS2740 Coulomb counter module setup.

It was observed while testing the DS2740 test case that the schematic design had a flaw. A pull up resistor R16 as illustrated in the reference design provided by MAXIM Figure 5.6 was not included, it was later and rectified by adding an appropriate pull up resistor and retested.

The test cases developed were simple but effective in identifying possible faults on the board.

REFERENCES

[1] ATMEL ATmega 128L AVR core microcontroller specifications. Website: www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

[2] A. H. Ansari "Hardware Development of an Embedded Wireless Evaluation Board", MS Thesis, *University of North Carolina - Charlotte*, Dec. 2005.

[3] A. Jutman "Selected Issues of Modeling, Verification and Testing of Digital Systems", PhD Thesis, *Tallinn University of Technology,* Tallinn, Oct. 2004.

[4] A. Wahba and D. Borrione "Connection errors location and correction in combinational Circuits", *Proc. European Design and Test Conference*, France, March 1997.

[5] B.K McElfresh "RF induction and analog junction techniques for finding opens", *IEEE International Test Conference. Proceedings*, 1997

[6] CC2420 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver specifications. Website: www.Chipcon.com/files/CC2420_Data_Sheet_1_2.pdf

[7] C.C. Lin, K.C. Chen, S.C. Chang and M. Cheng "Logic Synthesis for Engineering Change", *Proc. Design Automation Conference,* June 1995.

[8] DS2740 High-Precision Coulomb Counter specifications. Website: http://pdfserv.maxim-ic.com/en/ds/DS2740.pdf

[9] E. Callaway, P. Gorday, L. Hester, J.A.Gutierrez, M. Naeve, B. Heile, V.Bahl "Home Networking with IEEE 802.15.4: A Developing Standard for Low-Rate Wireless Personal Area Networks", *IEEE Communication Magazine,* August. 2002.

[10] J. A. Gutierrez "IEEE 802.15.4 WPAN-LR Task Group", *IEEE 802.15.4 Tutorial,* Jan. 2003.

[11] J. Catsoulis, *Designing Embedded Hardware*, O'Reilly Publication, 2002.

[12] JTAG interface "IEEE Std. 1149.1-1993, IEEE Standard Test Access Port and Boundary-Scan Architecture", IEEE, Inc., New York.

[13] K. Feldmann and J. Sturun "Closed loop quantity control in printed circuit Assembly", *IEEE Transactions on Components, Hybrids and Manufacturing Technology* Vol. 17, No.2, June 1994

[14] M. Galeey "Home Networking with ZigBee", *EE Times*, April 2004.

[15] M.L. Bushnell and V.D. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*, Kluwer Academic Publishers, 2000.

[16] Report "802.15.4 Market Could Grow 200% by 2009", *In-Stat Publication*, 2005.

[17] R. Goering "Productivity may stumble at 100 nm," *EE Times*, Sept. 2003.

[18] R. Rai "IEEE 802.15.4 Protocol Implementation and Measurement of Current Consumption", M.S Thesis, *University of North Carolina- Charlotte*, Dec. 2005.

[19] R. Ubar and D. Borrione "Generation of Tests for Localization of Single Gate Design Errors in Combinational Circuits Using the Stuck-at Fault Model", *Proc. 11th IEEE Brazilian Symposium on IC Design*, Brazil, Sept. 1998.

[20] W. C. Craig "Wireless Control that Simply Works"- *Communication Design Conference*, Oct. 2003.

[21] R.G.Wright, M. Zgol, D. Adebimpe and L.V Kirkland "Functional circuit board testing using nanoscale sensors*", IEEE Systems Readiness Technology Conference. Proceedings,* Sept 2003.

[22] Z. Peng and P. Eles "Testing of Digital Systems", Embedded System Laboratory (ESLAB), *Linköping University*. Website:http://www.glue.umd.edu/~seokjin/Testing%20Doc/Testing%20of%20Digitial%20Sys.pdf

APPENDIX A: SCHEMATICS

Figure A.1 Microcontroller ATmega128L interfaced with Coulomb counter DS2740

Figure A.2 RF Transceiver CC2420 interfaced with Coulomb Counter DS2740

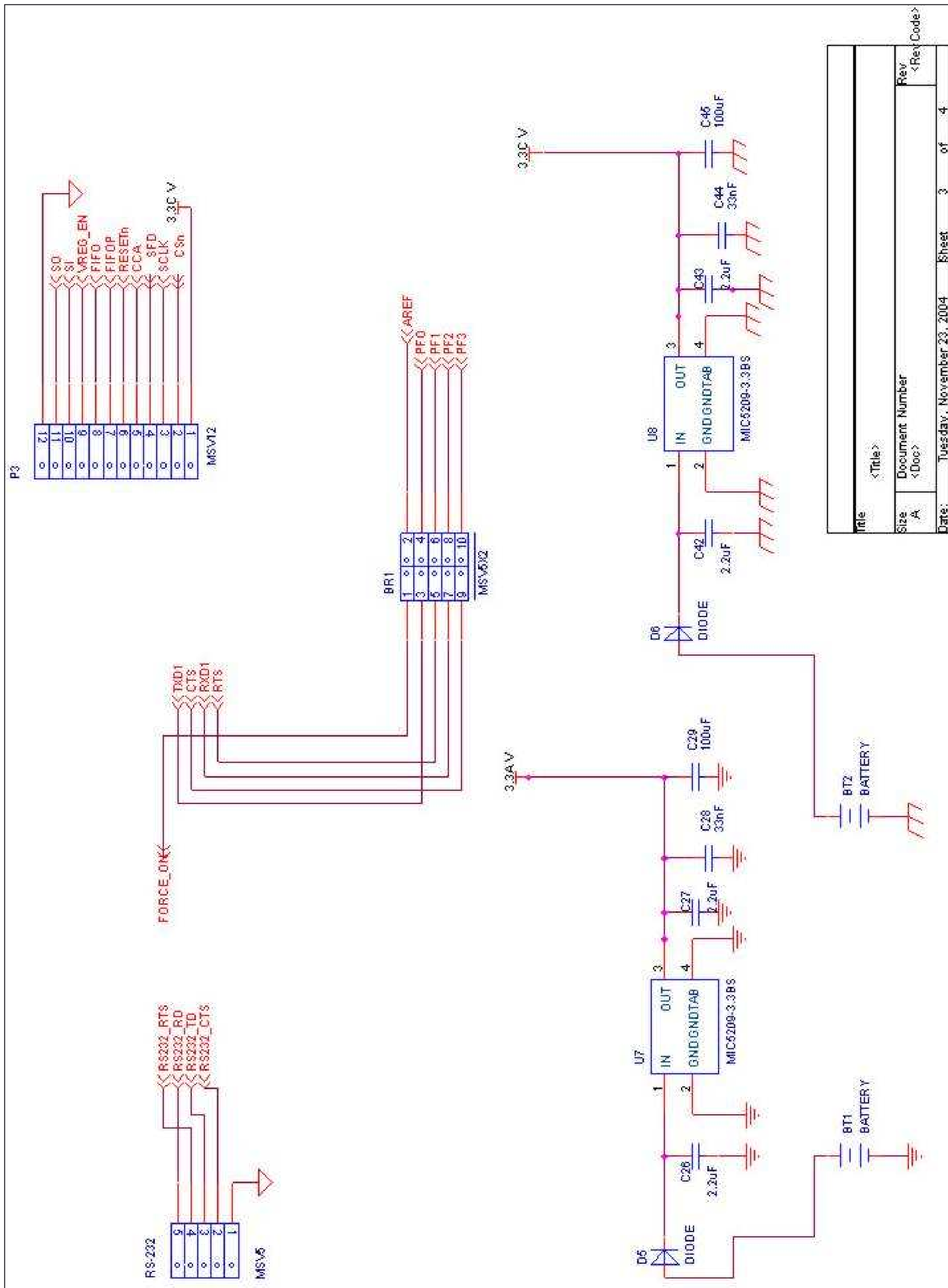


Figure A. 3 Power Supply for ATmega128L and CC2420 with some headers on the board

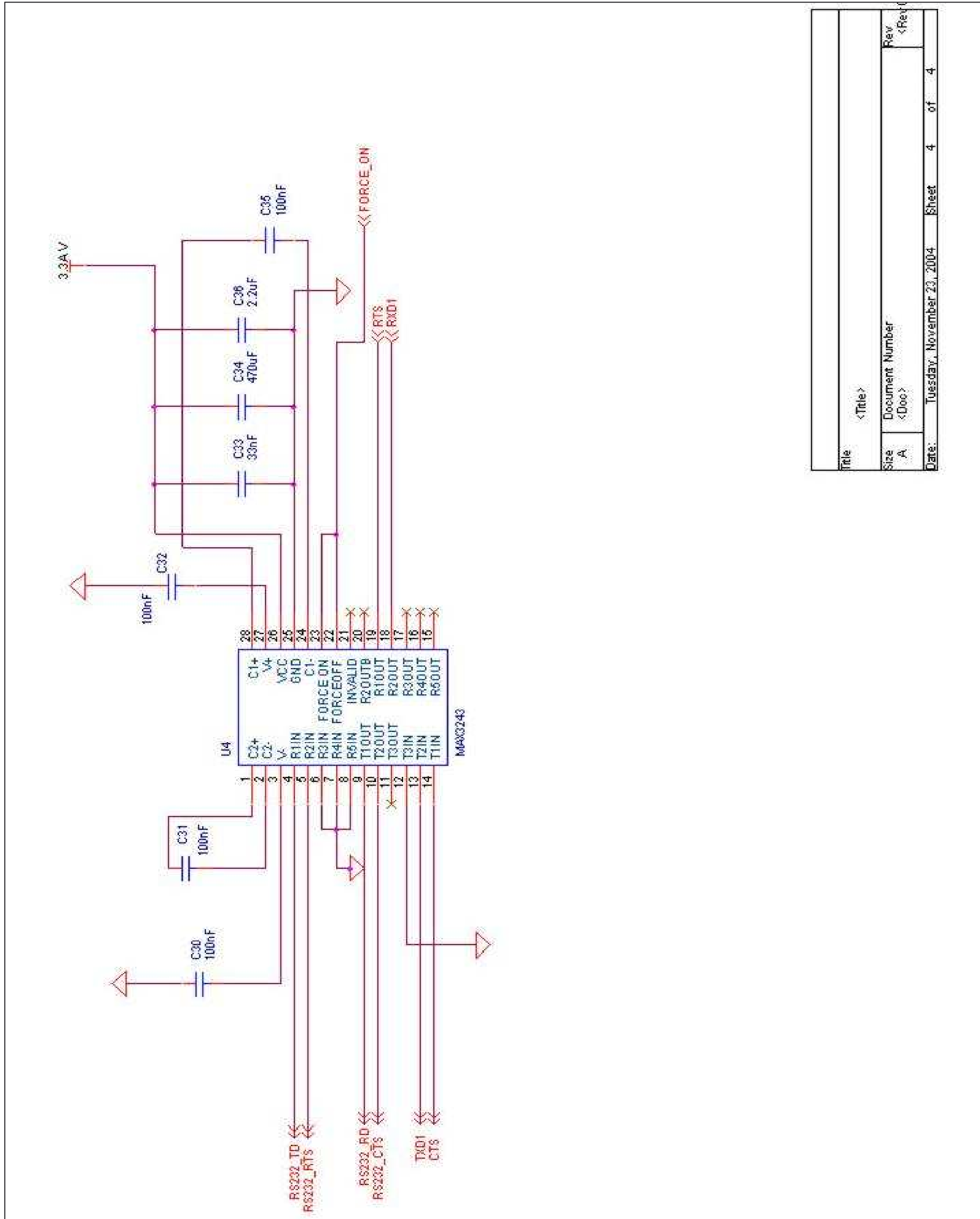


Figure A.4 MAX3243 RS-232 line Driver/Receiver

APPENDIX B: CODES

ISP.c

```
//************************************************************************
// File Name     : ISP.c
// Title         : Test case for ISP Header
// Target MCU  : Atmel AVR series
// 31-May-2005 Murari            Created the program
//************************************************************************

//----- ------------------------------ Include Files ---------------------------------------------------

#include <avr/io.h>                 // include I/O definitions (port names, pin names, etc)
#include "global.h"                 // include our global settings

int main(void)
{
        // All AVR processors have I/O ports which each contain up to 8
        // user-controllable pins.  From a hardware perspective, these I/O pins
        // are each an actual physical pin coming out of the processor chip.
        // The voltage on the pins can be sensed or controlled via software,
        // hence their designation as Input/Output pins.

        // While I/O pins are actual wires in the real world, they also exist
        // inside the processor as special memory locations called registers.
        // The software-controlled contents of these registers is what
        // determines the state and operation of I/O pins and I/O ports.

        // DDRx  - this register determines the direction (input/output) of the
        // pins on  port[x]
        //          A '0' bit in the DDR makes that port pin act as input
        //          A '1' bit in the DDR makes that port pin act as output

        // PORTx - this register contains the output state of the pins on port[x]
        //          A '0' bit makes the port pin output a LOW (~0V)
        //          A '1' bit makes the port pin output a HIGH (~5V)

        // PINx  - this register contains the input state of the pins on port[x]
        //          A '0' bit indicates that the port pin is LOW (at ~0V)
        //          A '1' bit indicates that the port pin is HIGH (at ~5V)

/************************** ISP Test Code ********************************/
        outb(DDRB, 0xFF);       // set all port B pins to output
        outb(PORTB, 0xFD);    // set PB1 as HIGH
        outb(DDRE, 0xFF);       // set all port B pins to output
        outb(PORTE, 0xFC);     // set PE0 and PE! as HIGH
}
/************************************************************************/
```

LED.c

```
//**************************************************************************
// File Name      : LED.c
// Title          : Test case for ISP Header
// Target MCU     : Atmel AVR series
// 31-May-2005 Murari           Created the program
//**************************************************************************


//----- Include Files ---------------------------------------------------------
#include <avr/io.h>              // include I/O definitions (port names, pin names, etc)
#include <global.h>              // include our global settings

int main(void)
{

        // All AVR processors have I/O ports which each contain up to 8
        // user-controllable pins.  From a hardware perspective, these I/O pins
        // are each an actual physical pin coming out of the processor chip.
        // The voltage on the pins can be sensed or controlled via software,
        // hence their designation as Input/Output pins.

        // While I/O pins are actual wires in the real world, they also exist
        // inside the processor as special memory locations called registers.
        // The software-controlled contents of these registers is what
        // determines the state and operation of I/O pins and I/O ports.

        // DDRx  - this register determines the direction (input/output) of the
        // pins on  port[x]
        //      A '0' bit in the DDR makes that port pin act as input
        //      A '1' bit in the DDR makes that port pin act as output

        // PORTx - this register contains the output state of the pins on port[x]
        //      A '0' bit makes the port pin output a LOW (~0V)
        //      A '1' bit makes the port pin output a HIGH (~5V)

        // PINx  - this register contains the input state of the pins on port[x]
        //      A '0' bit indicates that the port pin is LOW (at ~0V)
        //      A '1' bit indicates that the port pin is HIGH (at ~5V)
/*************************LED Test Code**************************************/

        outb(DDRG, 0xFF);       // set all port B pins to output
        outb(PORTG, 0xF8);      // set PB4-7 to HIGH and PG0-3 to LOW

}
/****************************************************************************/
```

LED_Timer.c

```
//************************************************************************
// File Name      : LED_Timer.c
// Title          : Test case for ISP Header
// Target MCU  : Atmel AVR series
// 31-May-2005 Murari              Created the program
//************************************************************************


//----- Include Files ---------------------------------------------------------
#include <avr/io.h>                  // include I/O definitions (port names, pin names, etc)
#include <avr/signal.h> // include "signal" names (interrupt names)
#include <avr/interrupt.h>         // include interrupt support
#include "global.h"                    // include our global settings
#include "timer128.h"                // include timer function library (timing, PWM, etc)

void init_port(void);
void init_timer(void);
void init_external_interrupt5(void);
int count = 0;

//----- Begin Code ------------------------------------------------------------
void main(void)
{
   init_port();
   init_timer();                             //Initialize Timer Interrupt
          for (;;);
}

void init_port(void)
{
                    DDRB = 0xFF;              // PORTB as output
                    DDRD = 0xFF;
                    PORTD = 0xFF;
                    PORTB = 0xFF;
}

/********  TIMER INTERRUPT INITIALIZATION ***********/
void init_timer(void)
{
          cbi(SREG, 7);
/*
          FOC0 WGM00 COM01 COM00 WGM01 CS02 CS01 CS00
            0         0            1           1        1          1        1      1
*/

          TCCR0 = 0x3F;
```

```
        TCNT0 = 0xFF;
        OCR0  = 0xFF;
        TIMSK = 0X02;                    // OCIE0 is enables
        sbi(SREG,7);
        sei();
}
/****************** INTERRUPT INITIALIZATION ****************/
void init_external_interrupt5(void)
{
        EIMSK = 0x00;  // Diable the INTERRUPT before writing into EICRA register
        cbi(SREG, 7);
        cbi(EICRA,6);     // ISC50 = 1; Logic level change triggers INTERRUPT
        cbi(EICRA,7);     //ISC51 = 0;

        EIMSK = 0x08;//Enable INTERRUPT
                                //Enable INTERRUPT enable bit
        sbi(SREG,7);
        sei();
}
/****************** TIMER 0 ISR **********************************/

SIGNAL(SIG_OUTPUT_COMPARE0)
                {
                count = count + 1;
                if (count%2)
                PORTB = 0x00;
                else PORTB = 0xFF;
                }
/**********************************************************************/
```

MSV12.c

```
//**********************************************************************
// File Name      : MSV12.c
// Title          : Test case for ISP Header
// Target MCU  : Atmel AVR series
// 31-May-2005 Murari              Created the program
//**********************************************************************

#include <avr/io.h>                // include I/O definitions (port names, pin names, etc)
#include "global.h"                // include our global settings

int main(void)
{
        // All AVR processors have I/O ports which each contain up to 8
        // user-controllable pins.  From a hardware perspective, these I/O pins
        // are each an actual physical pin coming out of the processor chip.
        // The voltage on the pins can be sensed or controlled via software,
        // hence their designation as Input/Output pins.

        // While I/O pins are actual wires in the real world, they also exist
        // inside the processor as special memory locations called registers.
```

```
        // The software-controlled contents of these registers is what
        // determines the state and operation of I/O pins and I/O ports.

        // DDRx  - this register determines the direction (input/output) of the
        // pins on  port[x]
        //        A '0' bit in the DDR makes that port pin act as input
        //        A '1' bit in the DDR makes that port pin act as output

        // PORTx - this register contains the output state of the pins on port[x]
        //        A '0' bit makes the port pin output a LOW (~0V)
        //        A '1' bit makes the port pin output a HIGH (~5V)

        // PINx  - this register contains the input state of the pins on port[x]
        //        A '0' bit indicates that the port pin is LOW (at ~0V)
        //        A '1' bit indicates that the port pin is HIGH (at ~5V)

/****************** MSV12 PORT HEADER Test Code ********************/

        outb(DDRB, 0xFF);       // set all port B pins to output
        outb(PORTB, 0x90);      // set PB4-7 to HIGH and PB0-3 to LOW
        outb(DDRD, 0xFF);       // set all port B pins to output
        outb(PORTD, 0xAC);      // set PB4-7 to HIGH and PB0-3 to LOW
}

/*****************************************************************/
```

MSV5.c

```
//*****************************************************************
// File Name     : MSV5.c
// Title         : Test case for ISP Header
// Target MCU    : Atmel AVR series
// 31-May-2005 Murari          Created the program
//*****************************************************************


#include <avr/io.h>             // include I/O definitions (port names, pin names, etc)
#include "global.h"             // include our global settings

//------------------------ Begin Code --------------------------------------//
int main(void)
{

/******** *************** MSV5 Test Code ****************************/

        outb(DDRE, 0xFF);       // set all port B pins to output
        outb(PORTE, 0x53);      // set PB4-7 to HIGH and PB0-3 to LOW

}
/*****************************************************************/
```

MSV5X2.c

```
//**********************************************************************
// File Name      : MSV5X2.c
// Title          : Test case for ISP Header
// Target MCU     : Atmel AVR series
// 31-May-2005 Murari            Created the program
//**********************************************************************

#include <avr/io.h>              // include I/O definitions (port names, pin names, etc)
#include "global.h"              // include our global settings

//------------------------- Begin Code ---------------------------------------//
int main(void)
{

        /************************** MSV5x2 Test Code*********************/

        outb(DDRF, 0xFF);        // set all port F pins to output
        outb(PORTF, 0xF0);       //
        outb(DDRD, 0xFF);        // set all port D pins to output
        outb(PORTD, 0x53);       //

}
/***********************************************************************/
```

Switch.c

```
//**********************************************************************
// File Name      : Switch.c
// Title          : Test case for ISP Header
// Target MCU     : Atmel AVR series
// 31-May-2005 Murari            Created the program
//**********************************************************************


//------------------------------------- Include Files -------------------------------------------/

#include <avr/io.h>              // include I/O definitions (port names, pin names, etc)
#include "global.h"              // include our global settings

int main(void)
{

/**************** SWITCHES USE PORTE ***********************/

        DDRB = 0xFF; // PORTB as output
        DDRD = 0x00; // PORTE as input
        PORTB = 0xFF;
        PORTD = 0x00;

                                        // Check for Switch press
                for(;;)                 //Forever
                {
```

```
                    a = inb(PIND);

                              if (PIND == 0xFD)
                              cbi(PORTB,1);
                              else sbi(PORTB,1);

                              if (PIND2==0)
                              cbi (PORTB, 2);
                              else

                              if (!PIND3)
                              {cbi (PORTB, 3);}
                              else sbi(PORT
                    }

}
/*****************************************************************************/
```

Switch_Interrupt.c

```
//*****************************************************************************
// File Name      : Switch_Interrupt.c

// When               Who                  Description of change
// -----------        -----------          -----------------------
// 02-Feb-2003  Murari             Created the program
//************************************************************************

//----- Include Files ---------------------------------------------------------
#include <avr/io.h>                   // include I/O definitions (port names, pin names, etc)
#include <avr/signal.h> // include "signal" names (interrupt names)
#include <avr/interrupt.h>          // include interrupt support

#include "global.h"                  // include our global settings
#include "timer128.h"               // include timer function library (timing, PWM, etc)

void init_port(void);
void init_external_interrupt5(void);
int count = 0;

//----- Begin Code ------------------------------------------------------------
void main(void)
{
   init_port();
   init_external_interrupt5();                              //Write a function to initialize interrupt
   for (;;);
}

void init_port(void)
{
                    DDRB = 0xFF; // PORTB as output
                    DDRD = 0xFF;
```

```
		//PORTD = 0xFF;
		//DDRD = 0x00;		// PORTE as input
		PORTD = 0xFF;
		PORTB = 0xFF;
}

/******************** INTERRUPT INITIALIZATION *****************/

void init_external_interrupt5(void)
{
	EIMSK = 0x00;  // Disable the INTERRUPT before writing into EICRA register
	cbi(SREG, 7);
	MCUCR = 0x02;
	cbi(EICRA,6);		//ISC50 = 1;	Logic level change triggers INTERRUPT
	cbi(EICRA,7);		//ISC51 = 0;


	sbi(EIFR, 3);
	EIMSK = 0x08;		//Enable INTERRUPT
				//Enable INTERRUPT enable bit
	sbi(SREG,7);
	sei();
}

/*************** EXTERNAL INTERRUPT ISR*************************/

SIGNAL(SIG_INTERRUPT3)
{
	count = count + 1;
	if (count%2)
	{
		cbi(PORTB,0);
		cbi(PORTB,1);
		cbi(PORTB,2);
	}
	else
	{
		sbi(PORTB,0);
		sbi(PORTB,1);
		sbi(PORTB,2);
	}
}
/*****************************************************************/
```

UART_Test.c

```
/*********************************************************************/
//File Name	: UART_Test.c

// When		Who
// -----------		-----------	-----------------------
// 02-Feb-2003		Murari		Modified
```

```
//**********************************************************************

#include <avr\io.h>
#include "USART1.h"

// Declare your global variables here
int main (void)
{
//-------------------------------------------------------------------
// USART0 initialization
// Communication Parameters: 8 Data, 1 Stop, No Parity
// USART0 Receiver: On
// USART0 Transmitter: On
// USART0 Mode: Asynchronous
// USART0 Baud rate: 4800
//-------------------------------------------------------------------
//UCSR0A=0x00;
//UCSR0B=0x18;
//UCSR0C=0x06;
//UBRR0H=0x00;


//UBRR0L=0x26; // For 3Mhz frequency 4800
//UBRR0L=0x67;   // For 8Mhz frequency 4800
//UBRR0L=0x33;   // For 16Mhz frequency 19200
//UBRR0L=0x33;  //for 4Mhz 4800

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

        USART0_Init(0x67 ); // Set the baud rate to 4800 bps using a 8MHz crystal

        for (;;)      /* Forever */
                {
                USART0_Transmit (USART0_Receive ()); // Echo the received character
                }

}//End of Main
/**********************************************************************/
```

```
//Filename:  UART1.c
#include <USART1.h>
/* Initialize UART */
void USART0_Init( unsigned int baudrate )
{
        /* Set the baud rate */
        UBRR0H = (unsigned char) (baudrate>>8);
        UBRR0L = (unsigned char) baudrate;

        /* Enable UART receiver and transmitter */
        UCSR0B = ( ( 1 << RXEN0 ) | ( 1 << TXEN0 ) );
```

```
        /* Set frame format: 8 N 1 */
        UCSR0C = (1<<UCSZ01)|(1<<UCSZ00); //For devices with Extended IO

        /* Set frame format: 8 data 2stop */ //For devices with Extended IO

//UCSR0C = (1<<USBS0)|(1<<UCSZ01)|(1<<UCSZ00);
//UCSR0C = (1<<URSEL)|(1<<USBS0)|(1<<UCSZ01)|(1<<UCSZ00);

/*
ATMEGA DATA SHEET page 192
UMSEL0     6  ==> Asynch 0, Synch 1
UPM01      5 ==> Parity Mode 0 Disable, 0 Reserved,1 Enb Even P,1 Enb odd P
UPM00      4 ==> Parity Mode 0 Disable, 1 Reserved, 0 Enb Even P,1 Enb odd P
USBS0      3 ==> Bit Setting 0 1 bit, 1  2bits
UCSZ01     2 ==> Data bit 11 for 8, 10 7bit
UCSZ00     1
UCPOL0     0 ==> Clock Polarity
*/
}
        /* Read and write functions */
        unsigned char USART0_Receive( void )
{       /* Wait for incomming data */
        while ( !(UCSR0A & (1<<RXC0)) );
        /* Return the data */
        return UDR0;    }

void USART0_Transmit( unsigned char data )
{       /* Wait for empty transmit buffer */
        while ( !(UCSR0A & (1<<UDRE0)) )    }

        /* Start transmittion */
        UDR0 = data;
}
/*****************************************************************************/
```

Filename: UART1.h

```
/* Includes */
#ifndef _USART1_H_
#define _USART1_H_
#include <io.h>

/* Prototypes */
        void USART0_Init (unsigned int baudrate );
        unsigned char USART0_Receive( void );
        void USART0_Transmit (unsigned char data );
#endif              // _USART1_H_
```

TEST FOR CC2420

```
/*************************************************************************/
// File Name      : RF_Led_Blink.c

// When             Who
// -----------                 -----------          -----------------------
// 02-Feb-2003              CHICPCON Support      Modified
//***********************************************************************

 /* This program demonstrates the use of the CC2420DB library, including the basic RF library.
The packet protocol being used is a small subset of the IEEE 802.15.4 standard. It uses an
802.15.4 MAC compatible frame format, but does not implement any other MAC
functions/mechanisms (e.g. CSMA-CA). The basic RF library can thus not be used to
communicate with compliant 802.15.4 networks. A pair of CC2420DBs running this program
will establish a point-to-point RF link on channel 26, using the following node addresses:
 * - PAN ID: 0x2420 (both nodes)
 * - Short address:
 * - 0x1234 if the joystick is moved in any direction at startup
 * - 0x5678 if the joystick button is pressed down at startup
 * PWM duty cycle on the receiving node. The other bytes are random (never initialized).
 *
 * LED indicators:
 *     - Red:    Transmission failed (acknowledgment not received)
 *     - Yellow: Transmission OK (acknowledgment received)
 *     - Orange: Remote controlled dimmer
  *   - Green:  Packet received
************************************************************************
 * Compiler: AVR-GCC
 * Target platform: CC2420DB (can easily be ported to other platforms)
 ***********************************************************************/

#include <include.h>


//----------------------------------------------------------------------------------------------------
// Basic RF transmission and reception structures
BASIC_RF_RX_INFO rfRxInfo;
BASIC_RF_TX_INFO rfTxInfo;
```

```
BYTE pTxBuffer[BASIC_RF_MAX_PAYLOAD_SIZE];
BYTE pRxBuffer[BASIC_RF_MAX_PAYLOAD_SIZE];
//-----------------------------------------------------------------------------------------------------
//  BASIC_RF_RX_INFO* basicRfReceivePacket(BASIC_RF_RX_INFO *pRRI)
//  DESCRIPTION:
/*This function is a part of the basic RF library, but must be declared by the application.  Once
the application has turned on the receiver, using basicRfReceiveOn(), all incoming packets will
be received by the FIFOP interrupt service routine. When finished, the ISR will call the
            basicRfReceivePacket() function. Please note that this function must return quickly, since
the next received packet will overwrite the active BASIC_RF_RX_INFO structure (pointed to by
pRRI).*/
//  ARGUMENTS:
//BASIC_RF_RX_INFO *pRRI
//The reception structure, which contains all relevant info about the received packet.
//  RETURN VALUE:
//  BASIC_RF_RX_INFO*
//  The pointer to the next BASIC_RF_RX_INFO structure to be used by the FIFOP ISR. //  If
there is only one buffer, then return pRRI.
//-----------------------------------------------------------------------------------------------------
BASIC_RF_RX_INFO* basicRfReceivePacket(BASIC_RF_RX_INFO *pRRI) {

   // Adjust the led brightness
   PWM0_SET_DUTY_CYCLE(pRRI->pPayload[0]);

   // Blink the green LED
   SET_GLED();
   halWait(10000);
   CLR_GLED();

   // Continue using the (one and only) reception structure
   return pRRI;

} // basicRfReceivePacket

//-----------------------------------------------------------------------------------------------------
//          void main (void)
//          DESCRIPTION:
//          Startup routine and main loop
//-----------------------------------------------------------------------------------------------------
void main (void) {
   UINT16 ledDutyCycle, dimmerDifference;
   UINT8 n;

   // Initalize ports for communication with CC2420 and other peripheral units
   PORT_INIT();
   SPI_INIT();

   // Initialize PWM0 with a period of CLK/1024
   PWM0_INIT(TIMER_CLK_DIV1024);

   // Initialize and enable the ADC for reading the pot meter
```

```
    ADC_INIT();
    ADC_SET_CHANNEL(ADC_INPUT_0_POT_METER);
        ADC_ENABLE();

    // Wait for the user to select node address, and initialize for basic RF operation
    while (TRUE) {
            if (JOYSTICK_CENTER_PRESSED()) {
                basicRfInit(&rfRxInfo, 26, 0x2420, 0x1234);
                rfTxInfo.destAddr = 0x5678;
                break;
            } else if (JOYSTICK_UP_PRESSED() || JOYSTICK_DOWN_PRESSED() ||
JOYSTICK_LEFT_PRESSED() || JOYSTICK_RIGHT_PRESSED()) {
                basicRfInit(&rfRxInfo, 26, 0x2420, 0x5678);
                rfTxInfo.destAddr = 0x1234;
                break;
            }
        }

    // Initalize common protocol parameters
    rfTxInfo.length = 10;
    rfTxInfo.ackRequest = TRUE;
    rfTxInfo.pPayload = pTxBuffer;
    rfRxInfo.pPayload = pRxBuffer;

    for (n = 0; n < 10; n++) {
      pTxBuffer[n] = n;
    }

    // Turn on RX mode
    basicRfReceiveOn();

        // The main loop:
        while (TRUE) {

      // Sample the pot meter value
      ADC_SAMPLE_SINGLE();
      ADC_GET_SAMPLE_8(ledDutyCycle);

/*If the dimmer value has changed by more than 1, then transmit the new value automatically*/
            // Transmit also when the S2 button is pressed
            dimmerDifference = (ledDutyCycle & 0xFF) - pTxBuffer[0];
            if ((ABS(dimmerDifference) > 2) || (JOYSTICK_CENTER_PRESSED())) {
              pTxBuffer[0] = ledDutyCycle;
            if (basicRfSendPacket(&rfTxInfo)) {

              // OK -> Blink the yellow LED
                  SET_YLED();
                  halWait(50000);
                  CLR_YLED();
                } else {
```

```
                                // No acknowledgment received -> Blink the red LED
                                SET_RLED();
                                halWait(50000);
                                CLR_RLED();
                            }
                    }
    }
} // main
```

Basic_RF.h

```
/*******************************************************************/
 The "Basic RF" library contains simple functions for packet transmission and reception with the
Chipcon CC2420 radio chip. The intention of this library is mainly to demonstrate how the
CC2420 is operated, and not to provide a complete and fully-functional packet protocol. The
protocol uses 802.15.4 MAC compliant data and acknowledgment packets, however it contains
only a small subset of the 802.15.4 standard:
 *     - Association, scanning, beacons is not implemented
 *     - No defined coordinator/device roles (peer-to-peer, all nodes are equal)
 *     - Waits for the channel to become ready, but does not check CCA twice (802.15.4 CSMA-
CA)
 *     - Does not retransmit packets
 *     - Can not communicate with other networks (using a different PAN identifier)
 *
 * INSTRUCTIONS:
 * Startup:
 *     1. Create a BASIC_RF_RX_INFO structure, and initialize the following members:
 *     - rfRxInfo.pPayload (must point to an array of at least
           BASIC_RF_MAX_PAYLOAD_SIZE bytes)
 *     2. Call basicRfInit() to initialize the packet protocol.
 * Transmission:
 *     1. Create a BASIC_RF_TX_INFO structure, and initialize the following members:
 *         - rfTxInfo.destAddr (the destination address, on the same PAN as you)
 *         - rfTxInfo.pPayload (the payload data to be transmitted to the other node)
 *         - rfTxInfo.length (the size od rfTxInfo.pPayload)
 *         - rfTxInfo.ackRequest (acknowledgment requested)
 *     2. Call basicRfSendPacket()
 *
 * Reception:
 *     1. Call basicRfReceiveOn() to enable packet reception
 *     2. When a packet arrives, the FIFOP interrupt will run, and will in turn call
 *        basicRfReceivePacket(), which must be defined by the application
 *     3. Call basicRfReceiveOff() to disable packet reception
 *
 * FRAME FORMATS:
 * Data packets:
 *     [Preambles (4)][SFD (1)][Length (1)][Frame control field (2)][Sequence number (1)][PAN
ID (2)]
 *     [Dest. address (2)][Source address (2)][Payload (Length - 2+1+2+2+2)][Frame check
sequence (2)]
```

```
 * Acknowledgment packets:
 *     [Preambles (4)][SFD (1)][Length = 5 (1)][Frame control field (2)][Sequence number (1)]
 *     [Frame check sequence (2)]
 * Compiler: AVR-GCC
 * Target platform: CC2420DB, CC2420 + any MCU with very few modifications required
 ***************************************************************************
 * Revision history:
 *  $Log: basic_rf.h,v $
 *  Revision 1.4  2004/07/26 11:26:15  mbr
 *  Modified BASIC_RF_ACK_DURATION & BASIC_RF_SYMBOL_DURATION
 *
 *  Revision 1.3  2004/03/30 14:58:45  mbr
 *  Release for web
***************************************************************************
#ifndef BASIC_RF_H
#define BASIC_RF_H

//*********** General constants
//------------------------------------------------------------------------------------------------------
// Constants concerned with the Basic RF packet format
// Packet overhead ((frame control field, sequence number, PAN ID, destination and
// source) + (footer))
// Note that the length byte itself is not included included in the packet length
#define BASIC_RF_PACKET_OVERHEAD_SIZE   ((2 + 1 + 2 + 2 + 2) + (2))
#define BASIC_RF_MAX_PAYLOAD_SIZE               (127 -
BASIC_RF_PACKET_OVERHEAD_SIZE)
#define BASIC_RF_ACK_PACKET_SIZE                5

// The time it takes for the acknowledgment packet to be received after the data packet
// has been transmitted
#define BASIC_RF_ACK_DURATION        (0.5 * 32 * 2 * ((4 + 1) + (1) + (2 + 1) + (2)))
#define BASIC_RF_SYMBOL_DURATION     (32 * 0.5)

// The length byte
#define BASIC_RF_LENGTH_MASK           0x7F

// Frame control field
#define BASIC_RF_FCF_NOACK             0x8841
#define BASIC_RF_FCF_ACK               0x8861
#define BASIC_RF_FCF_ACK_BM            0x0020
#define BASIC_RF_FCF_BM                (~BASIC_RF_FCF_ACK_BM)
#define BASIC_RF_ACK_FCF             0x0002

// Footer
#define BASIC_RF_CRC_OK_BM             0x80
//------------------------------------------------------------------------------------------------------
// *******************      Packet transmission          ********************/

// The data structure which is used to transmit packets
typedef struct {
   WORD destPanId;
```

```
          WORD destAddr;
          INT8 length;
   BYTE *pPayload;
          BOOL ackRequest;
} BASIC_RF_TX_INFO;

//-----------------------------------------------------------------------------------------------------
//  BYTE basicRfSendPacket(BASIC_RF_TX_INFO *pRTI)
//
//  DESCRIPTION:
//Transmits a packet using the IEEE 802.15.4 MAC data packet format with short addresses.
CCA ismeasured only once before backet transmission (not compliant with 802.15.4 CSMA-CA).
//   The function returns:
//- When pRTI->ackRequest is FALSE: After the transmission has begun
// (SFD gone high)
// - When pRTI->ackRequest is TRUE: After the acknowledgment has been
// received/declared missing.
// The acknowledgment is received through the FIFOP interrupt.
//  ARGUMENTS:
//  BASIC_RF_TX_INFO *pRTI
//  The transmission structure, which contains all relevant info about the packet.

//  RETURN VALUE:
//  BOOL
//  Successful transmission (acknowledgment received)
//-----------------------------------------------------------------------------------------------------
BOOL basicRfSendPacket(BASIC_RF_TX_INFO *pRTI);

// *************                Packet reception                *************************/
 // The receive struct:
typedef struct {
   BYTE seqNumber;
          WORD srcAddr;
          WORD srcPanId;
          INT8 length;
   BYTE *pPayload;
          BOOL ackRequest;
          INT8 rssi;
} BASIC_RF_RX_INFO;

//-----------------------------------------------------------------------------------------------------
//  void halRfReceiveOn(void)
//  DESCRIPTION:
//  Enables the CC2420 receiver and the FIFOP interrupt. When a packet is received
// through this interrupt, it will call halRfReceivePacket(...), which must be defined by the //
application
//-----------------------------------------------------------------------------------------------------
void basicRfReceiveOn(void);
//-----------------------------------------------------------------------------------------------------
//  void halRfReceiveOff(void)
//
```

```
//  DESCRIPTION:
//      Disables the CC2420 receiver and the FIFOP interrupt.
//----------------------------------------------------------------------------------------------------
void basicRfReceiveOff(void);


//----------------------------------------------------------------------------------------------------
//  SIGNAL(SIG_INTERRUPT0) - CC2420 FIFOP interrupt service routine
//  DESCRIPTION:
// When a packet has been completely received, this ISR will extract the data from the
// RX FIFO, put it into the active BASIC_RF_RX_INFO structure, and call
// basicRfReceivePacket() (defined by the application). FIFO overflow and illegally
// formatted packets is handled by this routine.
// Note: Packets are acknowledged automatically by CC2420 through the auto-
// acknowledgment feature.
//----------------------------------------------------------------------------------------------------
// SIGNAL(SIG_INTERRUPT0)


//----------------------------------------------------------------------------------------------------
//  BASIC_RF_RX_INFO* basicRfReceivePacket(BASIC_RF_RX_INFO *pRRI)
//  DESCRIPTION:
/*  This function is a part of the basic RF library, but must be declared by the application. Once
the application has turned on the receiver, using basicRfReceiveOn(), all incoming packets will
            be received by the FIFOP interrupt service routine. When finished, the ISR will call the
basicRfReceivePacket() function. Please note that this function must return quickly, since the
next received packet will overwrite the active BASIC_RF_RX_INFO structure (pointed to by
pRRI).*/

//  ARGUMENTS:
//  BASIC_RF_RX_INFO *pRRI
// The reception structure, which contains all relevant info about the received packet.
// RETURN VALUE:
// BASIC_RF_RX_INFO*
// The pointer to the next BASIC_RF_RX_INFO structure to be used by the FIFOP ISR. // If
there is only one buffer, then return pRRI.
//----------------------------------------------------------------------------------------------------
BASIC_RF_RX_INFO* basicRfReceivePacket(BASIC_RF_RX_INFO *pRRI);
/***************              Initialization              **************************/
// The RF settings structure:
typedef struct {
   BASIC_RF_RX_INFO *pRxInfo;
   UINT8 txSeqNumber;
   volatile BOOL ackReceived;
   WORD panId;
   WORD myAddr;
   BOOL receiveOn;
} BASIC_RF_SETTINGS;
extern volatile BASIC_RF_SETTINGS rfSettings;
//----------------------------------------------------------------------------------------------------
```

```
//----------------------------------------------------------------------------------------------------
// void basicRfInit(BASIC_RF_RX_INFO *pRRI, UINT8 channel, WORD panId, WORD
myAddr)
// DESCRIPTION:
/* Initializes CC2420 for radio communication via the basic RF library functions. Turns on the
voltage regulator, resets the CC2420, turns on the crystal oscillator, writes all necessary registers
and protocol addresses (for automatic address recognition). Note that the crystal oscillator will
remain on (forever).*/
// ARGUMENTS:
// BASIC_RF_RX_INFO *pRRI
// A pointer the BASIC_RF_RX_INFO data structure to be used during the first packet //
reception.
// The structure can be switched upon packet reception.
// UINT8 channel
// The RF channel to be used (11 = 2405 MHz to 26 = 2480 MHz)
// WORD panId
// The personal area network identification number
// WORD myAddr
/*The 16-bit short address which is used by this node. Must together with the PAN ID form a
unique 32-bit identifier to avoid addressing conflicts. Normally, in a 802.15.4 network, the short
address will be given to associated nodes by the PAN coordinator.*/
//----------------------------------------------------------------------------------------------------
void basicRfInit(BASIC_RF_RX_INFO *pRRI, UINT8 channel, WORD panId, WORD
myAddr);

#endif
```

Basic_RF_init.c

```
#include <include.h>
//----------------------------------------------------------------------------------------------------
// The RF settings structure is declared here, since we'll always need halRfInit()
// volatile BASIC_RF_SETTINGS rfSettings;
//----------------------------------------------------------------------------------------------------

//----------------------------------------------------------------------------------------------------
// void basicRfInit(BASIC_RF_RX_INFO *pRRI, UINT8 channel, WORD panId,
// WORD myAddr)
//
// DESCRIPTION:
/* Initializes CC2420 for radio communication via the basic RF library functions. Turns on the
voltage regulator, resets the CC2420, turns on the crystal oscillator, writes all necessary registers
and protocol addresses (for automatic address recognition). Note that the crystal oscillator will
remain on (forever).
// ARGUMENTS:
// BASIC_RF_RX_INFO *pRRI
// A pointer the BASIC_RF_RX_INFO data structure to be used during the first packet
// reception.
// The structure can be switched upon packet reception.
// UINT8 channel
```

```
// The RF channel to be used (11 = 2405 MHz to 26 = 2480 MHz)
// WORD panId
// The personal area network identification number
// WORD myAddr
// The 16-bit short address which is used by this node. Must together with the PAN ID
// form a unique 32-bit identifier to avoid addressing conflicts. Normally, in a 802.15.4
// network, the   short address will be given to associated nodes by the PAN coordinator.
//---------------------------------------------------------------------------------------------------
void basicRfInit(BASIC_RF_RX_INFO *pRRI, UINT8 channel, WORD panId, WORD
myAddr) {
   UINT8 n;

   // Make sure that the voltage regulator is on, and that the reset pin is inactive
   SET_VREG_ACTIVE();
   halWait(1000);
   SET_RESET_ACTIVE();
   halWait(1);
   SET_RESET_INACTIVE();
   halWait(5);

   // Initialize the FIFOP external interrupt
   FIFOP_INT_INIT();
   ENABLE_FIFOP_INT();

   // Turn off all interrupts while we're accessing the CC2420 registers
         DISABLE_GLOBAL_INT();


   // Register modifications
   FASTSPI_STROBE(CC2420_SXOSCON);

// Turn on automatic packet acknowledgment
FASTSPI_SETREG(CC2420_MDMCTRL0, 0x0AF2);

// Set the correlation threshold = 20
 FASTSPI_SETREG(CC2420_MDMCTRL1, 0x0500);

// Set the FIFOP threshold to maximum
FASTSPI_SETREG(CC2420_IOCFG0, 0x007F);

// Turn off "Security enable"
FASTSPI_SETREG(CC2420_SECCTRL0, 0x01C4);

   // Set the RF channel
   halRfSetChannel(channel);

   // Turn interrupts back on
   ENABLE_GLOBAL_INT();

         // Set the protocol configuration
         rfSettings.pRxInfo = pRRI;
```

```
            rfSettings.panId = panId;
            rfSettings.myAddr = myAddr;
            rfSettings.txSeqNumber = 0;
            rfSettings.receiveOn = FALSE;

            // Wait for the crystal oscillator to become stable
            halRfWaitForCrystalOscillator();

            // Write the short address and the PAN ID to the CC2420 RAM (requires that the //
XOSC is on and stable)
            DISABLE_GLOBAL_INT();
             FASTSPI_WRITE_RAM_LE(&myAddr, CC2420RAM_SHORTADDR, 2, n);
             FASTSPI_WRITE_RAM_LE(&panId, CC2420RAM_PANID, 2, n);
            ENABLE_GLOBAL_INT();

} // basicRfInit
```

RF_Receive.c

```
#include <include.h>
//-------------------------------------------------------------------------------------------------------
// void halRfReceiveOn(void)
//
// DESCRIPTION:
// Enables the CC2420 receiver and the FIFOP interrupt. When a packet is received
// through this interrupt, it will call halRfReceivePacket(...), which must be defined by the //
application
//-------------------------------------------------------------------------------------------------------
void basicRfReceiveOn(void) {
    rfSettings.receiveOn = TRUE;
            FASTSPI_STROBE(CC2420_SRXON);
            FASTSPI_STROBE(CC2420_SFLUSHRX);
    ENABLE_FIFOP_INT();
} // basicRfReceiveOn

//-------------------------------------------------------------------------------------------------------
//  void halRfReceiveOff(void)
//  DESCRIPTION:
//  Disables the CC2420 receiver and the FIFOP interrupt.
//-------------------------------------------------------------------------------------------------------
void basicRfReceiveOff(void) {
    rfSettings.receiveOn = FALSE;
    FASTSPI_STROBE(CC2420_SRFOFF);
    DISABLE_FIFOP_INT();
} // basicRfReceiveOff




//-------------------------------------------------------------------------------------------------------
//  SIGNAL(SIG_INTERRUPT0) - CC2420 FIFOP interrupt service routine
//  DESCRIPTION:
```

```
// When a packet has been completely received, this ISR will extract the data from the
// RX FIFO, put it into the active BASIC_RF_RX_INFO structure, and call
// basicRfReceivePacket() (defined by the application). FIFO overflow and illegally
// formatted packets is handled by this routine.
// Note: Packets are acknowledged automatically by CC2420 through the auto-
// acknowledgment feature.
//----------------------------------------------------------------------------------------------------
SIGNAL(SIG_INTERRUPT0) {
        WORD frameControlField;
        INT8 length;
        BYTE pFooter[2];

   // Clean up and exit in case of FIFO overflow, which is indicated by FIFOP = 1 and      FIFO =
0
        if((FIFOP_IS_1) && (!(FIFO_IS_1))) {
                 FASTSPI_STROBE(CC2420_SFLUSHRX);
                 FASTSPI_STROBE(CC2420_SFLUSHRX);
                return;
        }

        // Payload length
        FASTSPI_READ_FIFO_BYTE(length);
        length &= BASIC_RF_LENGTH_MASK; // Ignore MSB

   // Ignore the packet if the length is too short
   if (length < BASIC_RF_ACK_PACKET_SIZE) {
        FASTSPI_READ_FIFO_GARBAGE(length);

   // Otherwise, if the length is valid, then proceed with the rest of the packet
   } else {

      // Register the payload length
      rfSettings.pRxInfo->length = length - BASIC_RF_PACKET_OVERHEAD_SIZE;

      // Read the frame control field and the data sequence number
      FASTSPI_READ_FIFO_NO_WAIT((BYTE*) &frameControlField, 2);
      rfSettings.pRxInfo->ackRequest = !!(frameControlField & BASIC_RF_FCF_ACK_BM);
        FASTSPI_READ_FIFO_BYTE(rfSettings.pRxInfo->seqNumber);

                // Is this an acknowledgment packet?
        if ((length == BASIC_RF_ACK_PACKET_SIZE) && (frameControlField ==
BASIC_RF_ACK_FCF) && (rfSettings.pRxInfo->seqNumber == rfSettings.txSeqNumber)) {

                // Read the footer and check for CRC OK
                        FASTSPI_READ_FIFO_NO_WAIT((BYTE*) pFooter, 2);

// Indicate the successful ack reception (this flag is polled by the transmission routine)
                        if (pFooter[1] & BASIC_RF_CRC_OK_BM)
                rfSettings.ackReceived = TRUE;
                        } else if (length < BASIC_RF_PACKET_OVERHEAD_SIZE) {
                        FASTSPI_READ_FIFO_GARBAGE(length - 3);
```

```
                              return;

                    // Receive the rest of the packet
                    } else {
// Skip the destination PAN and address (that's taken care of by harware address
// recognition!)
                              FASTSPI_READ_FIFO_GARBAGE(4);

                              // Read the source address
FASTSPI_READ_FIFO_NO_WAIT((BYTE*) &rfSettings.pRxInfo->srcAddr, 2);
                              // Read the packet payload
FASTSPI_READ_FIFO_NO_WAIT(rfSettings.pRxInfo->pPayload, rfSettings.pRxInfo->length);

                              // Read the footer to get the RSSI value
                              FASTSPI_READ_FIFO_NO_WAIT((BYTE*) pFooter, 2);
                              rfSettings.pRxInfo->rssi = pFooter[0];

          // Notify the application about the received _data_ packet if the CRC is OK
                    if (((frameControlField & (BASIC_RF_FCF_BM)) ==
BASIC_RF_FCF_NOACK) && (pFooter[1] & BASIC_RF_CRC_OK_BM)) {
rfSettings.pRxInfo = basicRfReceivePacket(rfSettings.pRxInfo);
                              }
                    }
   }

} // SIGNAL(SIG_INTERRUPT0)
/*******************************************************************************/
```

Filename: Basic_Rf_Send_Packet.c

```
#include <include.h>
//-------------------------------------------------------------------------------------------------------
//  BYTE basicRfSendPacket(BASIC_RF_TX_INFO *pRTI)
//  DESCRIPTION:
Transmits a packet using the IEEE 802.15.4 MAC data packet format with short addresses. CCA
is measured only once before backet transmission (not compliant with 802.15.4 CSMA-CA).
          The function returns:
//- When pRTI->ackRequest is FALSE: After the transmission has begun (SFD gone
// high)
// - When pRTI->ackRequest is TRUE: After the acknowledgment has been
//  received/declared missing.
// The acknowledgment is received through the FIFOP interrupt.
//
//  ARGUMENTS:
//  BASIC_RF_TX_INFO *pRTI
//  The transmission structure, which contains all relevant info about the packet.
//
//  RETURN VALUE:
//  BOOL
//  Successful transmission (acknowledgment received)
```

```
//----------------------------------------------------------------------------------------------------
BOOL basicRfSendPacket(BASIC_RF_TX_INFO *pRTI) {
            WORD frameControlField;
            UINT8 packetLength;
            BOOL success;
           BYTE spiStatusByte;
   // Wait until the transceiver is idle
   while (FIFOP_IS_1 || SFD_IS_1);
// Turn off global interrupts to avoid interference on the SPI interface
   DISABLE_GLOBAL_INT();

            // Flush the TX FIFO just in case...
            FASTSPI_STROBE(CC2420_SFLUSHTX);

   // Turn on RX if necessary
   if (!rfSettings.receiveOn) FASTSPI_STROBE(CC2420_SRXON);

   // Wait for the RSSI value to become valid
   do {
      FASTSPI_UPD_STATUS(spiStatusByte);
   } while (!(spiStatusByte & BM(CC2420_RSSI_VALID)));

            // TX begins after the CCA check has passed
   do {
                      FASTSPI_STROBE(CC2420_STXONCCA);
                      FASTSPI_UPD_STATUS(spiStatusByte);
                      halWait(100);
   } while (!(spiStatusByte & BM(CC2420_TX_ACTIVE)));

// Write the packet to the TX FIFO (the FCS is appended automatically when AUTOCRC // is
enabled)
   packetLength = pRTI->length + BASIC_RF_PACKET_OVERHEAD_SIZE;
   FASTSPI_WRITE_FIFO((BYTE*)&packetLength, 1);
// Packet length
   frameControlField = pRTI->ackRequest ? BASIC_RF_FCF_ACK :
BASIC_RF_FCF_NOACK;
);
    // Frame control field
   FASTSPI_WRITE_FIFO((BYTE*) &frameControlField, 2

   // Sequence number
   FASTSPI_WRITE_FIFO((BYTE*) &rfSettings.txSeqNumber, 1);
   // Dest. PAN ID
FASTSPI_WRITE_FIFO((BYTE*) &rfSettings.panId, 2);
// Dest. address
FASTSPI_WRITE_FIFO((BYTE*) &pRTI->destAddr, 2);

// Source address
FASTSPI_WRITE_FIFO((BYTE*) &rfSettings.myAddr, 2);

// Payload
```

```
FASTSPI_WRITE_FIFO((BYTE*) pRTI->pPayload, pRTI->length);

// Wait for the transmission to begin before exiting (makes sure that this function cannot // be
called a second time, and thereby cancelling the first transmission (observe the
// FIFOP + SFD test above).
          while (!SFD_IS_1);
          success = TRUE;

// Turn interrupts back on
          ENABLE_GLOBAL_INT();

// Wait for the acknowledge to be received, if any
   if (pRTI->ackRequest) {
                    rfSettings.ackReceived = FALSE;

// Wait for the SFD to go low again
                    while (SFD_IS_1);

// We'll enter RX automatically, so just wait until we can be sure that the ack reception
// should have finished
// The timeout consists of a 12-symbol turnaround time, the ack packet duration, and a
// small margin
      halWait((12 * BASIC_RF_SYMBOL_DURATION) + (BASIC_RF_ACK_DURATION) +
(2 * BASIC_RF_SYMBOL_DURATION) + 100);

// If an acknowledgment has been received (by the FIFOP interrupt), the ackReceived
// flag should be set
                    success = rfSettings.ackReceived;
   }

          // Turn off the receiver if it should not continue to be enabled
          DISABLE_GLOBAL_INT();
          if (!rfSettings.receiveOn) FASTSPI_STROBE(CC2420_SRFOFF);
          ENABLE_GLOBAL_INT();

   // Increment the sequence number, and return the result
   rfSettings.txSeqNumber++;
   return success;
} // halRfSendPacket
```

```
/*******************************************************************************
  DS2740 Program To Read IEEE 802.15. Wierless Data Transmission.


//   by Rajan Rai <rajan_rai@yahoo.com>

  Desccription:
//- DS2740 and 1-Wire code is based on a sample from Peter Dannegger uses Peter Fleury's uart-
//library which is very portable between AVRs, added some functions in the uart-lib - CRC-check
//based on code from Colin O'Flynn access multiple sensors on multiple 1-Wire busses samples
//how to address every sensor in the bus by ROM-code independant of system-timers (more
//portable) but some (very short) delays used avr-libc inttypes no central include-file, parts of the
//code can be used as  "library" verbose output (different levels configureable) one-wire-bus can
//be changed at runtime if OW_ONE_BUS is not defined in onewire.h. There are still timing
//issues.
        Tests done with ATmega16 3,68MHz XTAL OK, , 8MHz intRC OK,
        4MHz intRC OK, 2MHz intRC OK, 1,84MHz XTAL OK, 1MHz intRC
        failed in runtime-configureable OW-Bus. All frequencies do
        work in OW_ONE_BUS-Mode.
*******************************************************************************/

#include <avr/io.h>
#include <avr/interrupt.h>
#include <string.h>

/* Hardware connection     End             */
/********************************************/

#include "uart.h"
#include "onewire.h"
#include "ds2740.h"

#include "delay.h"

#define SFD_POLLING
#define BAUD 9600

/********************************************/
/* Hardware connection     Start           */
```

```
// PC6 For One Wire Communication == DQ

// SFD PIN
#ifdef SFD_POLLING
#define BM(n) (1 << (n))
#define SFD_PIN 1                       // PA.1 - Input
#define SFD_PIN_HIGH() (PINA & BM(SFD_PIN))
#endif
#define MAXSENSORS 1

// Declare your global variables here
unsigned char rom_code[OW_ROMCODE_SIZE];
uint8_t search_sensors(void)
{
        uint8_t i;
        uint8_t id[OW_ROMCODE_SIZE];
        uint8_t diff, nSensors;

        uart_puts_P( "\n\rScanning Bus for DS2740\n\r" );

        nSensors = 0;

        for( diff = OW_SEARCH_FIRST;
                diff != OW_LAST_DEVICE && nSensors < MAXSENSORS ; )
        {
                DS2740_find_sensor( &diff, &id[0] );

                if( diff == OW_PRESENCE_ERR ) {
                        uart_puts_P( "No Sensor found\n\r" );
                        break;
                }

                if( diff == OW_DATA_ERR ) {
                        uart_puts_P( "Bus Error\n\r" );
                        break;
                }

                for (i=0;i<OW_ROMCODE_SIZE;i++)
                   rom_code[i]=id[i];

                nSensors++;
        }

        return nSensors;
}
#ifdef DS2740_VERBOSE

void printOneWireMeasurment(uint8_t startEnd){

  uint8_t nSensors;
```

```
 if (startEnd == 1)
     uart_puts_P( "\n\rVerbose output Start of Transmission\n\r" );
 else
     uart_puts_P( "\n\rVerbose output End of Transmission\n\r" );

 nSensors = search_sensors();
 uart_puti((int) nSensors);
 uart_puts_P( " DS2740 Sensor available:\n\r" );


 uart_puts_P("\n\r# Net Address : ");
 DS2740_read_meas_all_verbose();
 uart_puts_P( "--------------------------------------\n\r" );

}
#endif

int main( void )
{
        uint8_t nSensors;

        uart_init((UART_BAUD_SELECT((BAUD),F_OSC)));

        sei();

        nSensors = search_sensors();


        uart_puti((int) nSensors);
        uart_puts_P( " DS2740 Sensor available:\n\r" );



        #ifdef DS2740_VERBOSE

        while(1){
    if (SFD_PIN_HIGH()){
                printOneWireMeasurment(1); // Start of Transmission
                while(SFD_PIN_HIGH());
                printOneWireMeasurment(0); // End of Transmission
    }//if SFD High

   }//while 1
#endif

}

/***********************************************************************/
// Filename: onewire.h
#ifndef _1wire_h_
#define _1wire_h_
```

```
#include <inttypes.h>

/*******************************************/
/* Hardware connection                    */
/*******************************************/

/* Define OW_ONE_BUS if only one 1-Wire-Bus is used
   in the application -> shorter code.
   If not defined make sure to call ow_set_bus() before using
   a bus. Runtime bus-select increases code size by around 300
   bytes so use OW_ONE_BUS if possible */

//******Main Oscillator Frequency *********
//F_OSC = 1843200
//F_OSC = 2000000
//F_OSC = 3686400
//F_OSC = 4000000
#define F_OSC  8000000
//F_OSC   = 16000000
//Frequencies below 1,8MHz may only work in OW_ONE_BUS-Mode
//F_OSC =  1000000
#define OW_ONE_BUS

#ifdef OW_ONE_BUS
// One Wire Port Configuration
#define OW_PIN  PC6
#define OW_IN   PINC
#define OW_OUT  PORTC
#define OW_DDR  DDRC
#define OW_CONF_DELAYOFFSET 0

#else
#if F_OSC<1843200
 #warning | experimental multi-bus-mode is not tested for
 #warning | frequencies below 1,84MHz - use OW_ONE_WIRE or
 #warning | faster clock-source (i.e. internal 2MHz R/C-Osc.)
#endif
#define OW_CONF_CYCLESPERACCESS 13
#define OW_CONF_DELAYOFFSET ( (uint16_t)(
((OW_CONF_CYCLESPERACCESS)*1000000L) / F_OSC  ) )
#endif

/*******************************************/

// #define OW_SHORT_CIRCUIT  0x01

#define OW_MATCH_ROM     0x55
#define OW_SKIP_ROM          0xCC
#define OW_SEARCH_ROM    0xF0
```

```
#define OW_SEARCH_FIRST  0xFF               // start new search
#define OW_PRESENCE_ERR 0xFF
#define OW_DATA_ERR          0xFE
#define OW_LAST_DEVICE    0x00               // last device found
//                              0x01 ... 0x40: continue searching

// rom-code size including CRC
#define OW_ROMCODE_SIZE  8

extern uint8_t ow_reset(void);

extern uint8_t ow_bit_io( uint8_t b );
extern uint8_t ow_byte_wr( uint8_t b );
extern uint8_t ow_byte_rd( void );

extern uint8_t ow_rom_search( uint8_t diff, uint8_t *id );

extern void ow_command( uint8_t command, uint8_t *id );

extern void ow_parasite_enable(void);
extern void ow_parasite_disable(void);
extern uint8_t ow_input_pin_state(void);

#ifndef OW_ONE_BUS
extern void ow_set_bus(volatile uint8_t* in,
          volatile uint8_t* out,
          volatile uint8_t* ddr,
          uint8_t pin);
#endif

#endif


/************************************************************************/
/*
/*       Access Dallas 1-Wire Device with ATMEL AVRs
/*
/*             Author: Peter Dannegger
/*                         danni@specs.de
/*
/************************************************************************/

#include <avr/io.h>
#include <avr/interrupt.h>

#include "delay.h"
#include "onewire.h"
#ifdef OW_ONE_BUS

#define OW_GET_IN()   ( OW_IN & (1<<OW_PIN))
#define OW_OUT_LOW()  ( OW_OUT &= (~(1 << OW_PIN)) )
```

```
#define OW_OUT_HIGH() ( OW_OUT |= (1 << OW_PIN) )
#define OW_DIR_IN()   ( OW_DDR &= (~(1 << OW_PIN )) )
#define OW_DIR_OUT()  ( OW_DDR |= (1 << OW_PIN) )

#else

/* set bus-config with ow_set_bus() */
uint8_t OW_PIN_MASK;
volatile uint8_t* OW_IN;
volatile uint8_t* OW_OUT;
volatile uint8_t* OW_DDR;

#define OW_GET_IN()   ( *OW_IN & OW_PIN_MASK )
#define OW_OUT_LOW()  ( *OW_OUT &= (uint8_t) ~OW_PIN_MASK )
#define OW_OUT_HIGH() ( *OW_OUT |= (uint8_t)  OW_PIN_MASK )
#define OW_DIR_IN()   ( *OW_DDR &= (uint8_t) ~OW_PIN_MASK )
#define OW_DIR_OUT()  ( *OW_DDR |= (uint8_t)  OW_PIN_MASK )
//#define OW_PIN2 PD6
//#define OW_DIR_IN2()   ( *OW_DDR &= ~(1 << OW_PIN2 ) )

void ow_set_bus(volatile uint8_t* in,
	volatile uint8_t* out,
	volatile uint8_t* ddr,
	uint8_t pin)
{
	OW_DDR=ddr;
	OW_OUT=out;
	OW_IN=in;
	OW_PIN_MASK=(1<<pin);
	ow_reset();
}

#endif

uint8_t ow_input_pin_state()
{
	return OW_GET_IN();
}

void ow_parasite_enable(void)
{
   OW_OUT_HIGH();
	OW_DIR_OUT();
}

void ow_parasite_disable(void)
{
   OW_OUT_LOW();
	OW_DIR_IN();
}
```

```
uint8_t ow_reset(void)
{
	uint8_t err;
	uint8_t sreg;

	OW_OUT_LOW(); // disable internal pull-up (maybe on from parasite)
	OW_DIR_OUT(); // pull OW-Pin low for 480us

	delay_us(480);

	sreg=SREG;
	cli();

	// set Pin as input - wait for clients to pull low
	OW_DIR_IN(); // input

	delay_us(66);
	err = OW_GET_IN();		// no presence detect
	// nobody pulled to low, still high

	SREG=sreg; // sei()

	// after a delay the clients should release the line
	// and input-pin gets back to high due to pull-up-resistor
	delay_us(480-66);
	if( OW_GET_IN() == 0 )			// short circuit
		err = 1;

	return err;
}

/* Timing issue when using runtime-bus-selection (!OW_ONE_BUS):
   The master should sample at the end of the 15-slot after initiating
   the read-time-slot. The variable bus-settings need more
   cycles than the constant ones so the delays had to be shortened
   to achive a 15uS overall delay
   Setting/clearing a bit in I/O Register needs 1 cyle in OW_ONE_BUS
   but around 14 cyles in configureable bus (us-Delay is 4 cyles per uS) */
uint8_t ow_bit_io( uint8_t b )
{
	uint8_t sreg;
	sreg=SREG;
	cli();

	OW_DIR_OUT(); // drive bus low
	delay_us(1); // Recovery-Time wuffwuff was 1
	if ( b ) OW_DIR_IN(); // if bit is 1 set bus high (by ext. pull-up)

	// wuffwuff delay was 15uS-1 see comment above

	delay_us(15-1-OW_CONF_DELAYOFFSET);
```

```
        if( OW_GET_IN() == 0 ) b = 0;  // sample at end of read-timeslot
        delay_us(60-15);
        OW_DIR_IN();

        SREG=sreg; // sei();
        return b;
}

uint8_t ow_byte_wr( uint8_t b )
{
        uint8_t i = 8, j;
                do {
                        j = ow_bit_io( b & 1 );
                        b >>= 1;
                        if( j ) b |= 0x80;
                    } while( --i );
        return b;
}

uint8_t ow_byte_rd( void )
{
  // read by sending 0xff (a dontcare?)
  return ow_byte_wr( 0xFF );
}

uint8_t ow_rom_search( uint8_t diff, uint8_t *id )
{
        uint8_t i, j, next_diff;
        uint8_t b;
        if( ow_reset() ) return OW_PRESENCE_ERR;   // error, no device found
        ow_byte_wr( OW_SEARCH_ROM );                    // ROM search command
        next_diff = OW_LAST_DEVICE;                     // unchanged on last device
        i = OW_ROMCODE_SIZE * 8;                                // 8 bytes
                do {
                        j = 8;                                  // 8 bits
                do {
                        b = ow_bit_io( 1 );                     // read bit
                        if( ow_bit_io( 1 ) ) {                  // read complement bit
                                if( b )                         // 11
                                return OW_DATA_ERR;             // data error
                        }
                        else {
                                if( !b ) {                      // 00 = 2 devices
                                        if( diff > i || ((*id & 1) && diff != i) ) {
                                        b = 1;                  // now 1
                                        next_diff = i;          // Next pass 0
                                        }
                                }
                        }
                        ow_bit_io( b );                         // Write bit
                        *id >>= 1;
```

```
                if( b ) *id |= 0x80;                    // Store bit
                i--;
            } while( --j );
            id++;                                       // Next byte
        } while( i );
        return next_diff;                               // To continue search
}

void ow_command( uint8_t command, uint8_t *id )
{
        uint8_t i;
        ow_reset();
            if( id ) {
            ow_byte_wr( OW_MATCH_ROM );                 // To a single device
            i = OW_ROMCODE_SIZE;
                do {
                ow_byte_wr( *id );
                id++;
                } while( --i );
        }
            else {
            ow_byte_wr( OW_SKIP_ROM );                  // To all devices
            }
        ow_byte_wr( command );
}
```