The University of Sydney

ELEC3607 - Embedding Computing

Embedded System Assignment

An RFID Security System with Encrypted Wireless Transmission

Written By:

Nick Davies – 308165624

Adam Petrovic – 308165721

October, 2010

Contents

1 Problem Definition

This report is a detailed overview of our Embedded Systems Project, an RFID Security System with Encrypted Wireless Transmission

The problem we faced was trying to build a system whereby users can somehow identify themselves without the need to remember a PIN or Pass-code. We also wanted a system in which we could allow and deny access to certain users without the need to re-program the embedded device. As such, an administrator should also have the ability to add an allowed user remotely.

In addition to these problem requirements, the system needs a way of displaying a success / error notification for when something expected or unexpected happens within the system. This way the users and administrators of the system will know exactly when the system acts as expected or when it doesn't, why it didn't.

1.1 System's Solution to Problem

To solve the requirements of the Problem Description, we decided that users will authenticate using an RFID Token. RFID is a widely used technology that relies on the concept of a tag and reader. The tag is a token used to identify a person, object, animal and the reader can be used to identify a number of factors (time, location, identity) based off the unique code on the tag.

However for the purposes of this project, the focus is upon authenticating a person or user of the system. To clarify, the user is in possession of the RFID token, and the system itself contains the RFID reader. The next issue that we faced was the ability for this system to have allowed users added without having to physically re-program the device. This can be achieved through the use of a wireless module embedded into the system itself. This wireless module can then talk to a remote server with another wireless module, in which the authentication process commences and an appropriate signal is sent back. This design methodology also allows an administrator to simply connect to the server and add a new allowed user, without the device having to be reprogrammed or the system restarted.

Lastly, the issue of implementing a notification for when certain events occur was achieved using a series of LEDs. Each LED lights up in a manner corresponding to the various Messages the system provides (See Appendix (Section 8.2) for a detailed explanation).

2 System Design Overview

The following section will describe what is involved in implementation from the perspective of a Hardware and a Software view of the system. In these respective sections, each component of the system has been broken down and explained (in the form of schematic drawings for hardware or pseudo-code for software).

3 System Hardware Design

3.1 RFID Tags

The cards in which the system uses are basic RFID tags used for presence sensing. They work in the 125kHz Radio Frequency range. These tags come with a unique 32-bit ID and are not re-programmable, meaning that the IDs cannot change in attempt to bypass our security system. Also, as these tag IDs are a 32-bit number, this means there are just under 4.3 billion different RFID card combinations.

Technical Specifications:

- EM4001 ISO based RFID IC
- 125kHz Carrier
- 2kbps ASK
- Manchester encoding
- 32-bit unique ID
- 64-bit data stream [Header+ID+Data+Parity]

3.2 RFID Module

The ID-12 RFID Module is an RFID Reader. It contains a built-in antenna which is used for scanning the RFID tag.

RFID Module Features:

- 125kHz read frequency
- EM4001 64-bit RFID tag compatible
- 9600bps TTL and RS232 output
- Magnetic Stripe Emulation output
- 100mm Read Range (In practice this was more like 200mm)

To see how the RFID Module connects with the other components, see Appendix 3.8

The pinouts for the RFID Module can be found in Appendix 8.3

3.3 XBee Wireless Module

The XBee Wireless Module is a serial wireless transceiver. Our system utilises two XBee modules to establish the link between the PC (server) and the reader module. Details of the protocols and encryption schemes used are located in the Data Flow Section (Section 4.2.3 on Page 9).

The XBee Wireless Module features:

- 30 metres wireless range (Indoor)
- 100 metres wireless range (Outdoor with Line of Sight)
- RF Data-rate of 250,000bps
- Point-to-Point Network Topology
- UART Serial Interface

To see how the XBee Module is implemented on the Micro-Controller side, see Appendix 3.8

The pinouts for the XBee Wireless Module can be found in Appendix 8.4

3.4 LPC1343 Micro-Controller

The LPC1343 is an ARM Cortex-M3-based microcontroller that is the main component of the system. This micro-controller is used to consolidate all the pieces of hardware that are used in the system (aside from the XBee Wireless Module connected to a computer).
The LPC is the main controller of data flow in the system, this flow is detailed in section 4.2.3 on page 9

The LPC1343 Micro-Controller features:

- ARM Cortex-M3 processor running up to 72MHz
- Up to 32KB flash, up to 8KB SRAM
- UART, SPI Controller, I^2C-bus interface
- Up to 42 GPIO pins
- Integrated clock generation

The product data-sheet can be found at `http://ics.nxp.com/products/lpc1000/datasheet/lpc1311.lpc1313.lpc1342.lpc1343.pdf`.

3.5 Mode Switching: XBee Wireless and RFID Reader

3.5.1 The Problem

As can be seen from the above sections, our hardware design consists of a LPC1343 Microprocessor, a XBee wireless transceiver and an RFID reader. Both the XBee and the RFID reader must be connected to the LPC1343. However both of these modules provide a UART port as their only output data stream and the LPC only contains a single UART port. As a result, we needed to design the system in a way that allowed both devices to be physically connected to the LPC's UART port and allow for their data flows to be independent.

3.5.2 The Solution

The solution that our system implements for this problem is to use a 2:1 multiplexer to select the LPC's Rx pin, selecting from one of the two modules Tx lines. **Please Note:** this solution is only valid because we only ever wish to communicate with a single device at a time. We selected the 74LS257 TRI-STATE Quad 2-Data Selectors/Multiplexers as it suited our needs well. The main reason for this choice is due to the fact that it is a 5V TTL part, this choice is discussed in more detail in the System power supply section below.

3.6 System Power Supply

The system was powered by an external 5V power supply with a 3.3V linear regulator (LM7805 3.3V Regulator) to provide power for the lower voltage parts. The system is entirely independent of USB and does not rely on power coming from the LPC board in any way.

3.6.1 Interfacing 3.3V and 5V components

The mix of 3.3V and 5V parts was overcome in the following way. We chose the 5V multiplexer because as a TTL part it will accept 3.3V as HIGH and produce 5V HIGH out, this removes the need for any voltage shifting from 3.3V to 5V between the LPC and the RFID module (the XBee is wired directly to the LPC). Additionally due to the LPC having 5V tolerant input pins, the multiplexer can be directly connected to the LPC without the need for any level shifting.

3.7 Capacitors

3.7.1 Electrolytic $10\mu F$ Capacitors

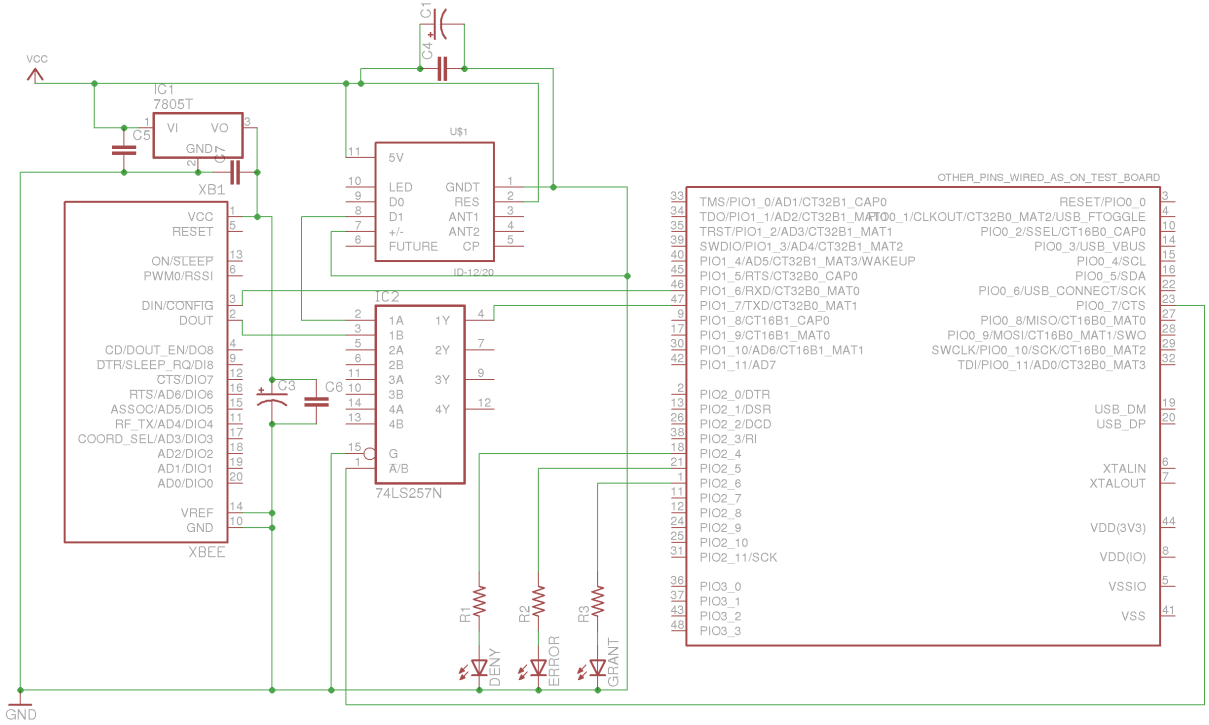
Electrolytic capacitors are placed between VCC and GND on the power supply, the XBee module and the RFID module, the reason for this is to compensate for the long ground and VCC lines on the breadboard. This provides a good reduction of low frequency noise and prevents voltage decay as well as to compensate for current spikes from the various components.

3.7.2 Ceramic $100nF$ Capacitors

Ceramic capacitors are placed very close to VCC and GND on the power supply, the XBee module and the RFID module, this is to prevent high frequency noise present in the system, from for example, the USB programmer.

3.8 Full Schematic Diagram

Below is a full schematic diagram of the system. This does not however show the wiring of the LPC beyond the GPIO pins used by the other components. This is not shown as it is neither pertinent or specific to our system.



4 System Software Design

4.1 Breakdown of Source Code

4.1.1 main

This module ties all the other modules together and controls the high level data flow for the system.

4.1.2 init

This module initialises all of the hardware devices, including GPIO pins, Timers, UART and then finally configure the XBee module using AT command mode.

4.1.3 mode

This module controls the mode that the system is currently in (RFID or Xbee). This controls the multiplexer through a GPIO pin. This is also wired to a LED which notifies the user as to when the system is ready to accept new cards or not given by the MODE LED.

4.1.4 rfid

The RFID module controls the decoding of the data from the module, checks and removes the start and stop sequences, converts the ASCII data to their HEX equivalent.

4.1.5 uart

The UART module send and receives any data that is transfered over the UART serial interface, this includes:

- The incoming data from the RFID module
- Data sent from the LPC to the XBee module
- Data received from the PC via the XBee module

This is where the UART interrupt handler resides in this module and is based upon two different variables

uart_max_read This variable holds the amount of data that is still needed to be read, every time a byte is read this value is decremented, this is used by other parts of the system to provide a blocking mechanism until enough data is read (a timeout is also enabled so it will not block indefinitely).

uart_dest_buffer A pointer to the destination, ie where the incoming data is to be written to. This buffer is not circular because of the previous variable, once enough data is read any additional (unexpected) data is discarded and the interrupt is cleared without the data being saved. This data reading is also not asynchronous and as such, there is no need for there to be continuous read/write without the buffer being fully read.

4.1.6 XBee

This module controls all of the XBee related uart controls, this includes the sending and receiving of packets including encoding decoding data into packets sending and receiving over UART and asks the crypto module to encrypt and decrypt packets that arrive and depart the system.

4.1.7 time

This module controls the delays and timeouts used throughout the system whilst it waits for various data flows to occur, this uses the LPC's built in 32-bit timer and allows for polling of whether or not a timeout has occurred.

This module also defines all of the timings in the system, for example the amount of time to wait for timeouts for each of the devices, as well as the timings for the LEDs and communications.

4.1.8 crypto

This module implements the encryption algorithm defined below in Section 4.2.4 on page 9. This encryption is used to encrypt all transmission to anything external to the embedded system. This provides protection for our system against various attacks, including packet spoofing and replay attacks.

4.1.9 errors

This modules defines all of the errors in the system that are used throughout the other modules.

4.1.10 IO

This module defines all of the information about Input and Output including pin information.

4.2 System Data Flows

After the system has successfully initialised it undertakes the following procedure (starting in RFID mode, assuming no errors have occurred):

1. Wait until an RFID tag is read by the RFID module, the system is notified by an interrupt by the UART module.
2. Check start, stop and CR/LF characters provided by the RFID (See Section 4.2.2 for more details).
3. Decode the RFID ID and checksum fields
4. Calculate and verify checksum of received data
5. Switch to XBee mode
6. Encode the request data into the wireless packet (defined in Section 4.2.3)
7. Encrypt entire datagram
8. Transmit datagram over UART to the XBee module
9. Wait and receive a response packet
10. Decrypt response datagram
11. Check random data (this can be caused by wrong encryption key, very delayed packet or a replay attack)
12. Calculate and validate data checksum
13. Perform the action requested in the response (grant/deny access)
14. Switch back to RFID mode
15. Return to step 1.

At any stage errors may occur, these are checked and the system will return to step 1 after displaying an error (A listing of Error codes can be found in Section 8.2 on page 12
This algorithm can be seen in main.c(page 14) including the error status checks.

4.2.1 Data Flow Diagram

Please see Appendix 8.1 for the Data Flow Diagram.

4.2.2 RFID Packet Details

The following defines the datagram that the RFID module provides to the system (numbers are in bytes):

0	1									11		13	14	15
STX(0x2)	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	C0	C1	LF	CR	ETX(0x3)

In the above diagram $D_0 \dots D_9$ denote the RFID unique ID and C_0 and C_1 denote checksum. Both of these fields are HEX data encoded in ASCII ie. if $D_0 = A = 65$ and $D_1 = F = 70$ then the first byte of the RFID card's ID would be equal to 0xAF.

4.2.3 Wireless Packet Definition

Below is the packet standard that we constructed for communications with the external server

0	1					6	7	8	10
R0	D0	D1	D2	D3	D4	R1	C0	R2	R3

$R_0 \dots R_3 = 4$ Bytes of random data spread throughout the packet
$D_0 \dots D_4 =$ The data being sent in the packet
$C_0 =$ The Checksum of the data being sent in the packet

The data is not encoded into ASCII as it is when it is received by the RFID module, this is to minimise transmission size.

The Random data ($R_0 \dots R_3$) is added to the packet to create randomness in the encrypted packet and to prevent replay attacks (more details in Section 4.2.4).

4.2.4 Encryption Algorithm

The data is encrypted before being sent over the wireless using a simple XOR algorithm we constructed. Random data is placed between every data byte as well as 4 bytes of random data at the end of the transmission. This is done because it prevents easy key exposure by decrypting known Start/Stop sequences. This way most of the key needs to be compromised before the full ID is revealed. Given a known RFID value you are not able to find the password because the random data remains unknown and hence cannot be decrypted. The checksum also adds increased security as the whole plain text string needs to be known before data can be spoofed. The random data also protects against replay attacks as the server must return the same random data it is given. This means that the encrypted stream will be completely different with every request, preventing a previous replay being replayed in response to a request.

Encryption/Decryption Algorithm Details:

$$P = Plain\ Text$$
$$K = Password$$
$$C = Cipher\ Text$$

$$Encryption:$$
$$C_0 = (P_0 \oplus K_0) \oplus K_1$$
$$C_i = (P_i \oplus C_{i-1}) \oplus K_{i+1}$$
$$Decryption:$$
$$P_0 = (C_0 \oplus K_1) \oplus K_0$$
$$P_i = (C_i \oplus K_{i+1}) \oplus C_{i-1}$$

4.3 System Output

The system has 1 on board LED and 3 external LEDs (one green, one orange and one red). These LEDs provide the systems communication with the user their behaviour is defined in the following way

4.3.1 LED allocation

- **Green**: Access Granted LED
- **Orange**: Access Denied LED
- **Red**: Error Notification LED
- **On-Board**: current MODE LED

4.3.2 Access Granted/Denied

Both the access granted and denied LEDs will blink 10 times with a 200ms on and 100ms off cycle when the respective response is received by the system.

4.3.3 Error Reporting

The error reporting method goes through the following procedure:

- Enable the error LED for 2 seconds
- Leave all LEDs off for 1 second
- Display the error code for the error that occurred (see Appendix 8.2 on page 12)
- Leave all LEDs off for 1 second
- Display the end of error led (signified by enabling the access granted light for 500ms)

4.3.4 Mode(System Busy) LED

The MODE pin was selected so that it shared its GPIO pin status with the on board LED, this means that when the on board LED is enabled the system is in XBee mode and as such it will act as a signal that the system is currently sending/receiving data, allowing the user to see when it is ready for a new card to be swiped (when the LED goes out).

5 System Testing

System testing was performed using the test_server.py python server instead of the standard server.py server (both of these files can be found in Section 8.6 starting on page 36). The standard server will read only a list of accepted cards and will never intentionally make mistakes that cause errors. The testing server on the other hand will read a list of cards for every error that it is able to cause. These errors include things such as: pretending that the checksum was not received correctly, send data with wrong checksum back, send not enough data, send too much data, get the random number wrong or simply do not reply at all. We could then swipe each of the cards relating to a given error and check that the correct error was displayed by the system.

5.1 Results

The system successfully reports all errors correctly and does not enter a state where it becomes non operational in any way. Overall our testing system was very comprehensive, and it allows for multiple errors to be found and corrected.

6 Possible Improvements

Since we have finished the project, we have contemplated adding one useful feature; The ability to add new authenticated users from the RFID reader side itself.

The theoretical process that this improvement would follow:

- Administrator places the RFID into a special 'save' mode
- System shows an appropriate message to confirm correct mode entry
- User scans card in order to add it to authorised cards list
- System confirms add process completion
- System returns back to normal read mode, where the card can be scanned and authenticated

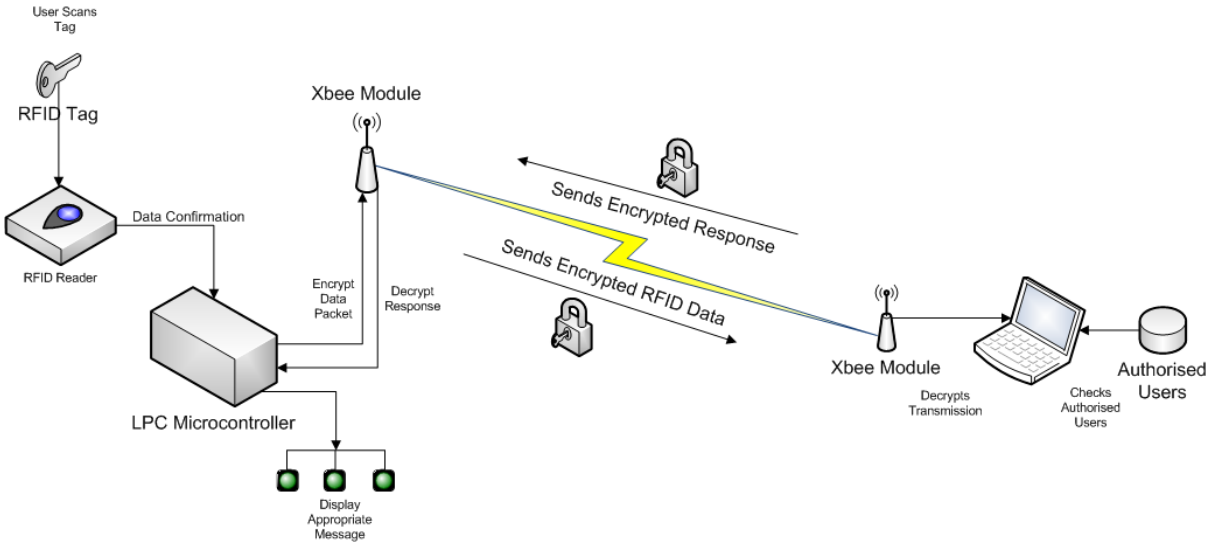
7 Conclusion

This design report has described the high-level components involved in creating this system, as well as how these components connect together on a circuitry level. Each component of the system has a provided schema / pinouts found in the Appendix, as well as code related to the running of these components.

Lastly, we have included a number of other supporting documents (Data Flow Diagram, Error Codes Table, Complete Source Code Listing). With these specifications we hope the reader can extrapolate enough information to understand the system at any level required; From understanding how RFID works to how to implement and RFID Reader.

8 Appendix

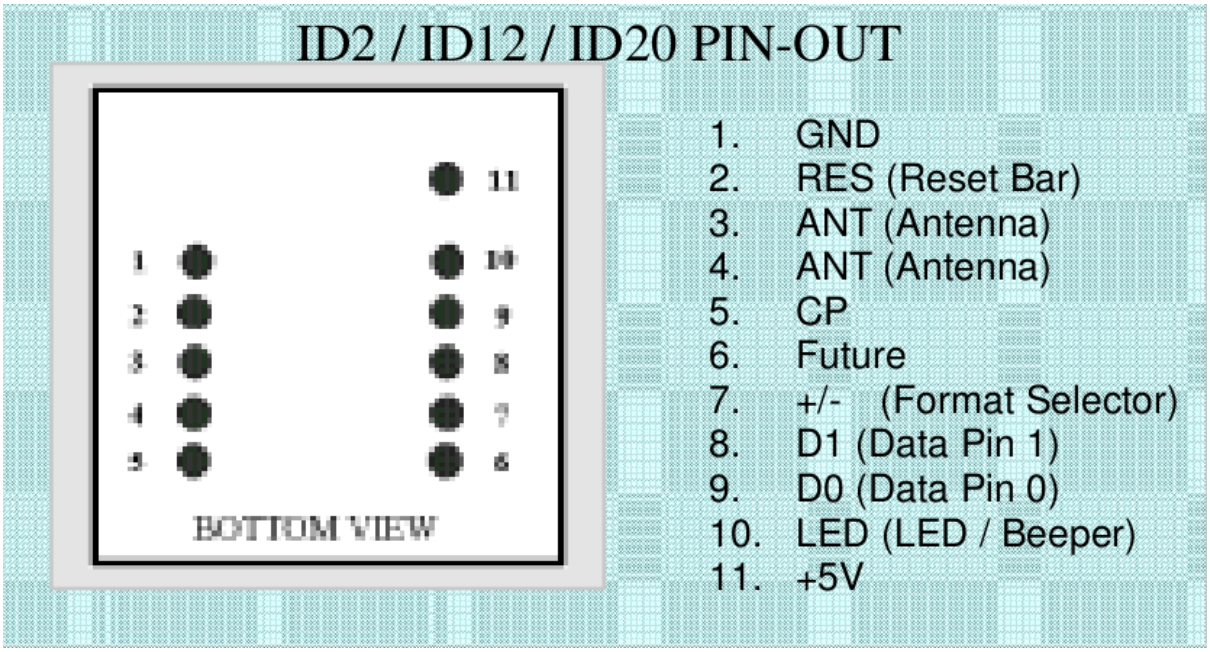
8.1 Data Flow Diagram



8.2 Error Codes

code	LED1	LED2	LED3	System Message
000	OFF	OFF	OFF	RFID Data Transmission Timeout
001	OFF	OFF	ON	Corrupt RFID Data Reading
010	OFF	ON	OFF	RFID Checksum Mismatch
011	OFF	ON	ON	UART Overflow Error
100	ON	OFF	OFF	XBee Initialisation Failure
101	ON	OFF	ON	XBee Data Transmission Timeout
110	ON	ON	OFF	XBee Response Timeout
111	ON	ON	ON	Random Data Response Mismatch

8.3 RFID pinouts



8.4 XBee pinouts

Pin #	Name	Direction	Description
1	VCC	-	Power supply
2	DOUT	Output	UART Data Out
3	DIN / $\overline{\text{CONFIG}}$	Input	UART Data In
4	DO8*	Output	Digital Output 8
5	$\overline{\text{RESET}}$	Input	Module Reset (reset pulse must be at least 200 ns)
6	PWM0 / RSSI	Output	PWM Output 0 / RX Signal Strength Indicator
7	PWM1	Output	PWM Output 1
8	[reserved]	-	Do not connect
9	$\overline{\text{DTR}}$ / SLEEP_RQ / DI8	Input	Pin Sleep Control Line or Digital Input 8
10	GND	-	Ground
11	AD4 / DIO4	Either	Analog Input 4 or Digital I/O 4
12	$\overline{\text{CTS}}$ / DIO7	Either	Clear-to-Send Flow Control or Digital I/O 7
13	ON / $\overline{\text{SLEEP}}$	Output	Module Status Indicator
14	VREF	Input	Voltage Reference for A/D Inputs
15	Associate / AD5 / DIO5	Either	Associated Indicator, Analog Input 5 or Digital I/O 5
16	$\overline{\text{RTS}}$ / AD6 / DIO6	Either	Request-to-Send Flow Control, Analog Input 6 or Digital I/O 6
17	AD3 / DIO3	Either	Analog Input 3 or Digital I/O 3
18	AD2 / DIO2	Either	Analog Input 2 or Digital I/O 2
19	AD1 / DIO1	Either	Analog Input 1 or Digital I/O 1
20	AD0 / DIO0	Either	Analog Input 0 or Digital I/O 0

8.5 Embedded Systems Source Code Listing

8.5.1 main.c

```
1  #include <stdio.h>
2
3  #include "errors.h"
4  #include "init.h"
5  #include "mode.h"
6  #include "rfid.h"
7  #include "time.h"
8  #include "xbee.h"
9
10 static void grant_access(void);
11 static void deny_access(void);
12 static void show_error(uint8_t e);
13
14 int main(void){
15
16   uint8_t status;
17   struct RFID_DATA rfid_id;
18
19   status = sys_init();
20
21   if (status == E_OK){
22     while(1){
23       set_mode(MODE_RFID);
24       status = read_rfid(&rfid_id);
25
26       if (status == E_OK){
27         set_mode(MODE_XBEE);
28
29         status = send_recieve_data(&rfid_id);
30         if (status == E_OK){
31           grant_access();
32         } else if (status == E_ACCESS_DENIED){
33           deny_access();
34         } else {
35           /* error sending/recieving responce from server
36              NOT including access denied */
37           show_error(status);
38         }
39       } else {
40         /* error reading RFID */
41         show_error(status);
42       }
43     }
44   } else {
45     while (1){
46       show_error(status);
47     }
48   }
49 }
50
51 static void grant_access(void){
52   uint8_t i;
53
54   LPC_GPIO2->DATA = 0;
55   for (i = 0; i != ACCESS_COUNT; i++){
56     LPC_GPIO2->DATA |= (1 << LED_OK_PIN);
```

```
57      delayms(ACCESS_ON_TIME);
58      LPC_GPIO2->DATA &= ~(1 << LED_OK_PIN);
59      delayms(ACCESS_OFF_TIME);
60    }
61
62  }
63
64  static void deny_access(void){
65    uint8_t i;
66
67    LPC_GPIO2->DATA = 0;
68    for (i = 0; i != ACCESS_COUNT; i++){
69        LPC_GPIO2->DATA |= (1 << LED_DENY_PIN);
70        delayms(ACCESS_ON_TIME);
71        LPC_GPIO2->DATA &= ~(1 << LED_DENY_PIN);
72        delayms(ACCESS_OFF_TIME);
73    }
74
75  }
76
77  static void show_error(uint8_t e){
78    LPC_GPIO2->DATA = 0;
79    LPC_GPIO2->DATA |= (1 << LED_ERROR_PIN);
80    delayms(ERROR_NOTIFY_TIME);
81
82    LPC_GPIO2->DATA = 0;
83    delayms(ERROR_NOTIFY_ERROR_GAP);
84
85    if (0x4 & e){
86      LPC_GPIO2->DATA |= (1 << LED_OK_PIN);
87    }
88    if (0x2 & e){
89      LPC_GPIO2->DATA |= (1 << LED_ERROR_PIN);
90    }
91    if (0x1 & e){
92      LPC_GPIO2->DATA |= (1 << LED_DENY_PIN);
93    }
94    delayms(ERROR_DISPLAY_TIME);
95    LPC_GPIO2->DATA = 0;
96    delayms(ERROR_END_GAP);
97    LPC_GPIO2->DATA |= (1 << LED_OK_PIN);
98    delayms(ERROR_END_OF_ERROR);
99    LPC_GPIO2->DATA = 0;
100 }
```

8.5.2 init.h

```
#ifndef init_GUARD
#define init_GUARD

/*public functions */
uint8_t sys_init(void);

#endif
```

8.5.3 init.c

```
#include "errors.h"
#include "init.h"
#include "uart.h"
#include "mode.h"
#include "time.h"
#include "xbee.h"

#include <stdlib.h>

/* private functions */

/* setup IOCONTRL and pin directions etc. */
static uint8_t IOInit(void);

/* setup basic uart functions */
static uint8_t UARTInit(void);

/* setup XBee module */
static uint8_t XBeeInit(void);


uint8_t sys_init(){

  uint32_t initc;
  uint32_t i = 0;
  uint32_t j = 0;
  uint8_t status;

  status = IOInit();
  if (status != E_OK){ return status; }

  initc = LPC_TMR32B1->TC;

  status = UARTInit();
  if (status != E_OK){ return status; }

  /* inaccurate pause to give clock time to start*/
  for(i = 0; i != CLOCK_INIT_PAUSE_LENGTH; i++){
    j++;
  }

  /* give other components time to power up and become ready for data*/
  delayms(EXTERNAL_COMPONENTS_BOOT_WAIT);

  status = XBeeInit();
  if (status != E_OK){ return status; }

```

```
48
49    LPC_GPIO2->DATA |= 0x70;
50    delayms(END_OF_INIT_LED_ON_TIME);
51    LPC_GPIO2->DATA &= ~(0x70);
52
53    srand(LPC_TMR32B1->TC - initc);
54
55    return E_OK;
56  }
57
58  static uint8_t IOInit(){
59
60    /* Clock Init */
61    // ACTUAL Frequency = 72000000hz = 72mhz
62    LPC_SYSCON->SYSAHBCLKCTRL |= (1<<10);
63    LPC_IOCON->PIO1_10 &= ~0x07;
64    LPC_IOCON->PIO1_10 |= 0x02;
65
66    /* MODE
67     * grant
68     * deny
69     * DIRECTION
70     */
71    LPC_GPIO0->DIR |= (1 << MODE_IO_PIN);
72    LPC_GPIO2->DIR |= 0x70;//(1 << LED_OK_PIN) & (1 << LED_ERROR_PIN) & (1 << ↩
          LED_DENY_PIN);
73    LPC_GPIO0->DATA = 0;
74    LPC_GPIO2->DATA = 0;
75
76    return E_OK;
77  }
78
79  static uint8_t UARTInit(void) {
80    uint32_t Fdiv;
81    uint32_t regVal;
82
83    NVIC_DisableIRQ(UART_IRQn);
84
85    LPC_IOCON->PIO1_6 &= ~0x07;    /*  UART I/O config */
86    LPC_IOCON->PIO1_6 |= 0x01;     /* UART RXD */
87    LPC_IOCON->PIO1_7 &= ~0x07;
88    LPC_IOCON->PIO1_7 |= 0x01;     /* UART TXD */
89    /* Enable UART clock */
90    LPC_SYSCON->SYSAHBCLKCTRL |= (1<<12);
91    LPC_SYSCON->UARTCLKDIV = 0x1;     /* divided by 1 */
92
93    LPC_UART->LCR = 0x83;             /* 8 bits, no Parity, 1 Stop bit */
94    regVal = LPC_SYSCON->UARTCLKDIV;
95    Fdiv = (((SystemCoreClock*LPC_SYSCON->SYSAHBCLKDIV)/regVal)/16)/UART_BAUDRATE↩
          ; /*baud rate */
96
97    LPC_UART->DLM = Fdiv / 256;
98    LPC_UART->DLL = Fdiv % 256;
99    LPC_UART->LCR = 0x03;    /* DLAB = 0 */
100   LPC_UART->FCR = 0x07;    /* Enable and reset TX and RX FIFO. */
101
102 #if UART_MODEM_ENABLE
103   LPC_UART->MCR = (1 << 4);
104 #endif
105
106   /* Read to clear the line status. */
```

```
107     regVal = LPC_UART->LSR;
108
109     /* Ensure a clean start, no data in either TX or RX FIFO. */
110     // CodeRed - added parentheses around comparison in operand of &
111     while (( LPC_UART->LSR & (LSR_THRE|LSR_TEMT)) != (LSR_THRE|LSR_TEMT) );
112     while ( LPC_UART->LSR & LSR_RDR )
113     {
114       regVal = LPC_UART->RBR; /* Dump data from RX FIFO */
115     }
116
117     /* Enable the UART Interrupt */
118     NVIC_EnableIRQ(UART_IRQn);
119
120     LPC_UART->IER = IER_RBR | IER_THRE | IER_RLS; /* Enable UART interrupt */
121
122     uart_max_read = 0;
123
124     return E_OK;
125   }
126
127   /* setup XBee module */
128   static uint8_t XBeeInit(){
129     uint8_t status = E_ACCESS_DENIED;
130
131     set_mode(MODE_XBEE);
132
133     status = enter_at_command_mode();
134     if (status != E_OK){ return status; }
135
136     status = send_at_command((uint8_t*)"ATID4214\r");
137     if (status != E_OK){ return status; }
138
139     status = send_at_command((uint8_t*)"ATMY01\r");
140     if (status != E_OK){ return status; }
141
142     status = send_at_command((uint8_t*)"ATDL10\r");
143     if (status != E_OK){ return status; }
144
145     status = send_at_command((uint8_t*)"ATCN\r");
146     if (status != E_OK){ return status; }
147
148     return E_OK;
149   }
```

8.5.4 mode.h

```
#ifndef mode_GUARD
#define mode_GUARD

#ifdef __USE_CMSIS
#include "LPC13xx.h"
#endif

/* Modes */
#define MODE_RFID 0
#define MODE_XBEE    1

/* mode pins */
#define MODE_IO_PORT LPC_GPIO0->DATA
#define MODE_IO_PIN 7

uint8_t get_mode();
void set_mode(uint8_t mode);

uint8_t current_mode;

#endif
```

8.5.5 mode.c

```

#include "mode.h"
#include "IO.h"

uint8_t get_mode(){
  return current_mode;
}

void set_mode(uint8_t mode){

  MODE_IO_PORT &= (0 << MODE_IO_PIN);
  MODE_IO_PORT |= (mode << MODE_IO_PIN);
  current_mode = mode;

}
```

8.5.6 rfid.h

```
#ifndef rfid_GUARD
#define rfid_GUARD

#define RFID_STX 0x02
#define RFID_ETX 0x03

#define RFID_ASCII_DATA_LENGTH 10
#define RFID_FULL_READ_LENGTH 16
/* should be 1/2 of the value above */
#define RFID_HEX_DATA_LENGTH 5

struct RFID_DATA_READ {
  /* Start Symbol */
  uint8_t STX;

  /* Data ASCII */
  uint8_t data[RFID_ASCII_DATA_LENGTH];
  /* Checksum */
  uint8_t checksum[2];

  /* CRAP */
    uint8_t CR;
    uint8_t LF;
  /* Stop Symbol */
  uint8_t ETX;
};

struct RFID_DATA {
  /* Data HEX */
  uint8_t data[RFID_HEX_DATA_LENGTH];
  /* Checksum */
  uint8_t checksum;
};

uint8_t read_rfid(struct RFID_DATA* rfid_data);
uint8_t verify_rfid_checksum(struct RFID_DATA in);

#endif
```

8.5.7 rfid.c

```
#include "errors.h"
#include "rfid.h"
#include "uart.h"
#include "time.h"

/* Private Functions */
static uint8_t read_raw_rfid(struct RFID_DATA_READ* target);
static uint8_t condence_rfid_data(struct RFID_DATA_READ* in, struct RFID_DATA *↩
    out);
static uint8_t ASCII_to_HEX(uint8_t ascii, uint8_t* hex);

/* Public Functions */
uint8_t read_rfid(struct RFID_DATA* rfid_data){
  struct RFID_DATA_READ input_data;
  uint8_t status;

```

```
16    status = read_raw_rfid(&input_data);
17    if( status != E_OK){ return status; }
18
19    status = condence_rfid_data(&input_data,rfid_data);
20    if( status != E_OK){ return status; }
21
22    status = verify_rfid_checksum(*rfid_data);
23    if( status != E_OK){ return status; }
24
25    return E_OK;
26  }
27
28  /* Private Functions */
29  static uint8_t read_raw_rfid(struct RFID_DATA_READ* target){
30
31    uint8_t data[16];
32
33    volatile uint32_t i,full_length;
34
35    uart_dest_buffer = data;
36    uart_max_read = RFID_FULL_READ_LENGTH;
37
38    /* wait for start of transmission */
39    full_length = RFID_FULL_READ_LENGTH;
40    while( uart_max_read == full_length){ i++; }
41
42    /* Timeout if all data isnt read */
43    setTimeOut(RFID_READ_TIMEOUT);
44    while( !timedOut() && uart_max_read ){ i++; }
45
46    if ( timedOut() ){
47      return E_RFID_TIMEOUT;
48    }
49
50    target->STX = data[0];
51    for (i = 0; i != RFID_ASCII_DATA_LENGTH; i++){
52      target->data[i] = data[i+1];
53    }
54    target->checksum[0] = data[11];
55    target->checksum[1] = data[12];
56
57    target->CR = data[13];
58    target->LF = data[14];
59    target->ETX = data[15];
60
61    return E_OK;
62  }
63
64  static uint8_t condence_rfid_data(struct RFID_DATA_READ* in, struct RFID_DATA *↩
        out){
65
66    uint8_t i;
67    uint8_t tmp_L;
68    uint8_t tmp_H;
69
70    /* Verify start/stop symbols and \r\n characters
71      as a form of error detection */
72    if (in -> STX != RFID_STX ||
73      in -> ETX != RFID_ETX ||
74      in -> CR  != '\r' ||
75      in -> LF  != '\n' ){
```

```
76
77      return E_RFID_CORRUPT;
78    }
79
80    for(i = 0;i != RFID_ASCII_DATA_LENGTH; i += 2){
81      if (ASCII_to_HEX(in->data[i],&tmp_H) != E_OK ||
82        ASCII_to_HEX(in->data[i+1],&tmp_L) != E_OK){
83
84        return E_RFID_CORRUPT;
85      }
86        out -> data[i/2] = (tmp_H << 4) | tmp_L;
87    }
88    if (ASCII_to_HEX(in->checksum[0],&tmp_H) != E_OK ||
89      ASCII_to_HEX(in->checksum[1],&tmp_L) != E_OK){
90
91      return E_RFID_CORRUPT;
92    }
93    out -> checksum = (tmp_H << 4) | tmp_L;
94    return E_OK;
95  }
96
97  uint8_t verify_rfid_checksum(struct RFID_DATA in){
98    uint8_t i;
99    uint8_t checksum = in.data[0];
100
101   for(i = 1;i != RFID_HEX_DATA_LENGTH; i++ ){
102     checksum ^= in.data[i];
103   }
104
105   if( checksum == in.checksum ){
106     return E_OK;
107   }
108   return E_RFID_CHECKSUM;
109 }
110
111 static uint8_t ASCII_to_HEX(uint8_t ascii, uint8_t* hex){
112
113   if (ascii <= '9' && ascii >= '0'){
114       *hex = ascii - '0';
115     return E_OK;
116   }
117
118   if (ascii <= 'F' && ascii >= 'A'){
119       *hex = ascii - 'A' + 10;
120     return E_OK;
121   }
122
123   return E_RFID_CORRUPT;
124 }
```

8.5.8 uart.h

```
#ifndef uart_GUARD
#define uart_GUARD

#ifdef __USE_CMSIS
#include "LPC13xx.h"
#endif

#include "time.h"

/* UART SETTINGS */
#define UART_BAUDRATE 9600
#define UART_MODEM_ENABLE 0

/* INTERRUPT MODES */
#define UART_TX_INTERRUPT 1
#define UART_RX_INTERRUPT 1

/* INTERRUPT ENABLE VALUES */
#define IER_RBR    0x01
#define IER_THRE   0x02
#define IER_RLS    0x04

/* INTERRUPT TYPE VALUES */
#define IIR_PEND   0x01
#define IIR_RLS    0x03
#define IIR_RDA    0x02
#define IIR_CTI    0x06
#define IIR_THRE   0x01

/* PIN INFORMATION */
#define LSR_RDR    0x01
#define LSR_OE     0x02
#define LSR_PE     0x04
#define LSR_FE     0x08
#define LSR_BI     0x10
#define LSR_THRE   0x20
#define LSR_TEMT   0x40
#define LSR_RXFE   0x80

uint8_t * uart_dest_buffer;
uint32_t uart_max_read;
uint32_t UARTStatus;
uint8_t  UARTTxEmpty;

/* UART FUNCTIONS */

void UART_IRQHandler(void);
void UARTSend(uint8_t *BufferPtr, uint32_t Length);

#endif
```

8.5.9 uart.c

```
#include "uart.h"
#include "errors.h"

void UART_IRQHandler(void)
```

```
5  {
6    uint8_t IIRValue, LSRValue;
7    uint8_t Dummy;
8
9    IIRValue = LPC_UART->IIR;
10
11   IIRValue >>= 1;      /* skip pending bit in IIR */
12   IIRValue &= 0x07;      /* check bit 1~3, interrupt identification */
13   if (IIRValue == IIR_RLS)  /* Receive Line Status */
14   {
15     LSRValue = LPC_UART->LSR;
16     /* Receive Line Status */
17     if (LSRValue & (LSR_OE | LSR_PE | LSR_FE | LSR_RXFE | LSR_BI))
18     {
19       /* There are errors or break interrupt */
20       /* Read LSR will clear the interrupt */
21       UARTStatus = LSRValue;
22       Dummy = LPC_UART->RBR;  /* Dummy read on RX to clear
23                       interrupt, then bail out */
24       return;
25     }
26     if (LSRValue & LSR_RDR) /* Receive Data Ready */
27     {
28       if (uart_max_read){
29         *uart_dest_buffer++ = LPC_UART->RBR;
30         uart_max_read--;
31       } else {
32         UARTStatus = E_UART_OVERFLOW;
33         UARTStatus = LSRValue;
34         Dummy = LPC_UART->RBR;  /* Dummy read on RX to clear
35                         interrupt, then bail out */
36         return;
37       }
38     }
39   }
40   else if (IIRValue == IIR_RDA) /* Receive Data Available */
41   {
42     if (uart_max_read){
43       *uart_dest_buffer++ = LPC_UART->RBR;
44       uart_max_read--;
45     } else {
46       UARTStatus = E_UART_OVERFLOW;
47       UARTStatus = LSRValue;
48       Dummy = LPC_UART->RBR;  /* Dummy read on RX to clear
49                       interrupt, then bail out */
50       return;
51     }
52   }
53   else if (IIRValue == IIR_CTI) /* Character timeout indicator */
54   {
55     /* Character Time-out indicator */
56     UARTStatus |= 0x100;     /* Bit 9 as the CTI error */
57   }
58   else if (IIRValue == IIR_THRE)  /* THRE, transmit holding register empty */
59   {
60     /* THRE interrupt */
61     LSRValue = LPC_UART->LSR;   /* Check status in the LSR to see if
62                         valid data in U0THR or not */
63     if (LSRValue & LSR_THRE)
64     {
65       UARTTxEmpty = 1;
```

```
66      }
67      else
68      {
69        UARTTxEmpty = 0;
70      }
71    }
72    return;
73  }
74
75  void UARTSend(uint8_t *BufferPtr, uint32_t Length){
76
77    while ( Length != 0 ) {
78      /* THRE status, contain valid data */
79      while ( !(LPC_UART->LSR & LSR_THRE) );
80      LPC_UART->THR = *BufferPtr;
81        BufferPtr++;
82        Length--;
83    }
84    return;
85  }
```

8.5.10 xbee.h

```
#ifndef xbee_GUARD
#define xbee_GUARD

#ifdef __USE_CMSIS
#include "LPC13xx.h"
#endif

#include "rfid.h"

/* packet details */
#define XB_DATAGRAM_SIZE 10

/* public functions */
uint8_t send_recieve_data(struct RFID_DATA * send);

uint8_t send_at_command(uint8_t* cmd);
uint8_t enter_at_command_mode(void);

#endif
```

8.5.11 xbee.c

```
#include "errors.h"
#include "time.h"
#include "xbee.h"
#include "crypto.h"
#include "rfid.h"
#include "uart.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* private functions */
static uint8_t send_data(struct RFID_DATA * send,uint32_t * random);
static uint8_t recieve_data(uint32_t * random);

static uint8_t encode_random(uint8_t * full_datagram, struct RFID_DATA * ↩
    datastream,uint8_t * random);
static uint8_t decode_random(uint8_t * full_datagram, struct RFID_DATA * ↩
    data_out,uint8_t * random);


uint8_t send_recieve_data(struct RFID_DATA * send){

  uint8_t status;
  uint32_t random;

  random = rand();

  status = send_data(send,&random);
  if (status != E_OK){ return status; }

  return recieve_data(&random);

}

```

```
34  uint8_t send_at_command(uint8_t* cmd){
35    volatile uint32_t i, responce = 0;
36
37    uart_dest_buffer = (uint8_t *)&responce;
38    uart_max_read = 3;
39
40    UARTSend(cmd,strlen( (char*)cmd));
41    setTimeOut(XBEE_COMMAND_TIMEOUT);
42    while( !timedOut() && uart_max_read ){ i++; }
43    if ( timedOut() ){
44        return E_XBEE_INIT_FAILURE;
45    }
46
47    if (responce != 0x0d4b4f){
48        return E_XBEE_INIT_FAILURE;
49    }
50
51    return E_OK;
52  }
53
54  uint8_t enter_at_command_mode(void){
55
56    volatile uint32_t i, responce = 0;
57
58    uart_dest_buffer = (uint8_t *)&responce;
59    uart_max_read = 3;
60
61    delayms(XBEE_GUARD_TIMES);
62
63    UARTSend((uint8_t*)"+++",3);
64
65    delayms(XBEE_GUARD_TIMES);
66
67    setTimeOut(XBEE_COMMAND_TIMEOUT);
68
69    while( !timedOut() && uart_max_read ){ i++; }
70    if ( timedOut() ){
71        return E_XBEE_INIT_FAILURE;
72    }
73
74    if (responce != 0x0d4b4f){
75      return E_XBEE_INIT_FAILURE;
76    }
77
78    return E_OK;
79
80  }
81
82  static uint8_t send_data(struct RFID_DATA * send,uint32_t * random){
83    uint8_t status;
84    uint8_t full_datagram[XB_DATAGRAM_SIZE];
85
86
87
88    //uint8_t interlace_random(uint8_t * full_datagram, uint8_t * datastream, ↩
          uint32_t size){
89    status = encode_random(full_datagram,send,(uint8_t*)random);
90
91    if (status != E_OK){ return status; }
92
93  #if CRYPTO_ENABLE_ENCRYPTION
```

```
94    status = encrypt(CRYPTO_PASSWORD,full_datagram,XB_DATAGRAM_SIZE);
95    if (status != E_OK){ return status; }
96  #endif
97
98    UARTSend(full_datagram,XB_DATAGRAM_SIZE);
99
100   return E_OK;
101
102 }
103
104 static uint8_t recieve_data(uint32_t * random){
105
106   /* data data input */
107   uint8_t full_datagram[XB_DATAGRAM_SIZE];
108   struct RFID_DATA read;
109   uint8_t status;
110
111   /* Busy waiting counters */
112   volatile uint32_t i,full_length;
113
114   /* We are waiting for XB_DATAGRAM_SIZE Bytes input */
115   uart_dest_buffer = full_datagram;
116   uart_max_read = XB_DATAGRAM_SIZE;
117
118   /* wait for start of transmission */
119   full_length = XB_DATAGRAM_SIZE;
120   setTimeOut(XBEE_RESPONCE_TIMEOUT);
121
122   /* wait for any data */
123   while( !timedOut() && uart_max_read == full_length){ i++; }
124   /* If transmission didnt start in time, timeout */
125   if ( timedOut() ){
126       return E_XBEE_RESPONCE_TIMEOUT;
127   }
128
129   /* Timeout if all data isnt read */
130   setTimeOut(XBEE_INCOMPLETE_TIMEOUT);
131   while( !timedOut() && uart_max_read ){ i++; }
132   /* If we did not receive as much data as we wanted */
133   if ( timedOut() ){
134     return E_XBEE_DATA_TIMEOUT;
135   }
136
137 #if CRYPTO_ENABLE_ENCRYPTION
138   status = decrypt(CRYPTO_PASSWORD,full_datagram,XB_DATAGRAM_SIZE);
139   if (status != E_OK){ return status; }
140 #endif
141
142   decode_random(full_datagram,&read,(uint8_t*)&i);
143   if (i != *random){
144     //printf("%p %p\n",i,*random);
145     return E_RANDOM_MISSMATCH;
146   }
147
148   status = verify_rfid_checksum(read);
149   if (status != E_OK){ return status; }
150
151   return (read.data[0] * 8 +
152       read.data[1] * 4 +
153       read.data[2] * 2 +
154       read.data[3]
```

```
155         );
156 }
157
158 /*
159  * r_i = Random data
160  * d_i = data
161  * c_i = checksum
162  *
163  * SIZE:  1  1  1  1  1  1  1  1  1  1 = 10
164  * DATA: r0 d0 d1 d2 d3 d4 r1 c1 r2 r3
165  *
166  */
167 static uint8_t encode_random(uint8_t * full_datagram, struct RFID_DATA * ↩
        datastream,uint8_t * random){
168
169   uint8_t * data = datastream->data;
170
171   *(full_datagram++) = *(random++);
172   *(full_datagram++) = *(data++);
173   *(full_datagram++) = *(data++);
174   *(full_datagram++) = *(data++);
175   *(full_datagram++) = *(data++);
176   *(full_datagram++) = *(data++);
177   *(full_datagram++) = *(random++);
178   *(full_datagram++) = datastream->checksum;
179   *(full_datagram++) = *(random++);
180   *(full_datagram++) = *(random++);
181
182   return E_OK;
183 }
184
185 static uint8_t decode_random(uint8_t * full_datagram, struct RFID_DATA * ↩
        data_out,uint8_t * random){
186
187   uint8_t * data = data_out->data;
188
189   *(random++) = *(full_datagram++);
190   *(data++) = *(full_datagram++);
191   *(data++) = *(full_datagram++);
192   *(data++) = *(full_datagram++);
193   *(data++) = *(full_datagram++);
194   *(data++) = *(full_datagram++);
195   *(random++) = *(full_datagram++);
196   data_out->checksum = *(full_datagram++);
197   *(random++) = *(full_datagram++);
198   *(random++) = *(full_datagram++);
199
200   return E_OK;
201 }
```

8.5.12 time.h

```
#ifndef time_GUARD
#define time_GUARD
#ifdef __USE_CMSIS
#include "LPC13xx.h"
#endif

/* Timeings for transmissions */
#define RFID_READ_TIMEOUT 200
#define XBEE_RESPONCE_TIMEOUT 1000
#define XBEE_INCOMPLETE_TIMEOUT 300

/* Led Timings */
#define ACCESS_ON_TIME 200
#define ACCESS_OFF_TIME 100
#define ACCESS_COUNT 10

/* error LEDS */
#define ERROR_NOTIFY_TIME 2000
#define ERROR_NOTIFY_ERROR_GAP 1000
#define ERROR_DISPLAY_TIME 2000
#define ERROR_END_GAP 1000
#define ERROR_END_OF_ERROR 500

/* Times for Inits */
#define XBEE_GUARD_TIMES 1000
/* Normal response time <15ms so this is easily enough time */
#define XBEE_COMMAND_TIMEOUT 50
#define END_OF_INIT_LED_ON_TIME 1000

/* This is not an accurate delay and hence is not in ms */
#define CLOCK_INIT_PAUSE_LENGTH 10000
#define EXTERNAL_COMPONENTS_BOOT_WAIT 500



/* functions */
void setTimeOut(uint32_t ms);

uint8_t timedOut(void);

void delayms(uint32_t ms);
#endif
```

8.5.13 time.c

```

#include "time.h"

void setTimeOut(uint32_t ms){

  LPC_TMR32B1->TCR = 0x2;
  LPC_TMR32B1->MCR = (1 << 2);
  LPC_TMR32B1->MR0 = ms * ((SystemCoreClock/LPC_SYSCON->SYSAHBCLKDIV) / 1000);
  LPC_TMR32B1->TCR = 0x1;

}

```

```
13 uint8_t timedOut(void){
14   return !(LPC_TMR32B1->TCR & 0x1);
15 }
16
17 void delayms(uint32_t ms){
18   setTimeOut(ms);
19
20   volatile uint32_t i = 0;
21   while(!timedOut()){
22     i++;
23   }
24 }
```

8.5.14 crypto.h

```
#ifndef crypto_GUARD
#define crypto_GUARD

#include <stdint.h>


#define CRYPTO_ENABLE_ENCRYPTION 1

/* should be quite long so that its definatly longer than the datagram */
#define CRYPTO_PASSWORD (uint8_t*)"lUBe43hOnEzin50IdeAl"

uint8_t encrypt(uint8_t * pw, uint8_t * data,uint32_t len);
uint8_t decrypt(uint8_t * pw, uint8_t * data,uint32_t len);

#endif
```

8.5.15 crypto.c

```
#include "errors.h"
#include "crypto.h"

#include <stdio.h>

/*
  Encryption algorithm will be as follows:
    pw[] as password bytes
    P[] as plain text bytes
    C[] as ciper bytes
  PW is 1 element longer than P and C will be as long as P

  C[0] = (P[0] ^ PW[0]) ^ PW[1]

  for ( i = 1 to i = len(P) )
    C[i] = (P[i] ^ P[i-1]) ^ PW[i+1]

  Done
*/

uint8_t encrypt(uint8_t * pw, uint8_t * data,uint32_t len){
  uint32_t i;

  /* C[0] = (P[0] XOR PW[0]) XOR PW[1] */
  *data = ( (*data) ^ (*pw)) ^ *(pw + 1);
  pw++;

  for( i = 1; i != len; i++){
    data ++;pw ++;
    *data = ( (*data) ^ *(data-1) ) ^ *pw;
  }

  return E_OK;

}

uint8_t decrypt(uint8_t * pw, uint8_t * data,uint32_t len){

  uint32_t i;
```

```
40    uint8_t prev_crypt;
41    uint8_t tmp;
42
43    prev_crypt = *data;
44    *data = ( *data ^ *(pw + 1)) ^ *(pw);
45
46    pw++;
47
48    for( i = 1; i != len; i++){
49        data++; pw++;
50      tmp = *data;
51      *data = (*data ^ *pw) ^ prev_crypt;
52      prev_crypt = tmp;
53
54    }
55    return E_OK;
56  }
```

8.5.16 errors.h

```
1  #ifndef errors_GUARD
2  #define errors_GUARD
3
4  #include "LPC13xx.h"
5
6  #define E_OK              0x8
7  #define E_ACCESS_DENIED       0xa
8
9  /* rfid errors */
10 #define E_RFID_TIMEOUT        0x0 // 000
11 #define E_RFID_CORRUPT        0x1 // 001
12 #define E_RFID_CHECKSUM       0x2 // 010
13
14 /* UART errors */
15 #define E_UART_OVERFLOW       0x3 // 011
16
17 /* XBEE wireless errors */
18 #define E_XBEE_INIT_FAILURE     0x4 // 100
19 #define E_XBEE_DATA_TIMEOUT     0x5 // 101
20 #define E_XBEE_RESPONCE_TIMEOUT 0x6 // 110
21 #define E_RANDOM_MISSMATCH      0x7 // 111
22
23 /* LEDS */
24 #define LED_OK_PIN    6
25 #define LED_ERROR_PIN    5
26 #define LED_DENY_PIN    4
27
28
29 #endif
```

8.5.17 IO.h

```
1 #ifndef IO_GUARD
2 #define IO_GUARD
3
4 #define MODE_DATA_ADDRESS 0x50000004
5
6
7
8 #endif
```

8.6 External Server Source Code Listing

8.6.1 Primary Server

```
import serial
import time

def open_port():
  return serial.Serial('/dev/ttyUSB0',9600)

def enter_command_mode(ser):
  time.sleep(1)
  ser.write('+++')
  time.sleep(1)

  if(ser.read(3) != "OK\r"):
    return False
  return True

def at_command(cmd,ser):
  ser.write(cmd + "\r")
  if(ser.read(3) != "OK\r"):
    return False
  return True


def setup_device():
  at_commands = ["ATID4214","ATDL01","ATMY10","ATCN"]

  ser = open_port()
  if (enter_command_mode(ser)):

    for command in at_commands:

      if ( not at_command(command,ser) ):
        print "fail on:", command
        return None

    print "setup OK"
    return ser

  else:
    print "failed to enter command mode"
    return None

def encrypt(pw,data):
  data = list(data)
  data[0] = chr(ord(data[0]) ^ ord(pw[0]) ^ ord(pw[1]))
  for i in xrange(1,len(data)):
    data[i] = chr(ord(data[i]) ^ ord(data[i-1]) ^ ord(pw[i+1]))

  return "".join(data)

def decrypt(pw,data):
  data = list(data)

  prev_crypt = ord(pw[0])
  for (i,item) in enumerate(data):
    data[i] = chr( ord(data[i]) ^ ord(pw[i+1]) ^ prev_crypt)
    prev_crypt = ord(item)
```

```
57
58    return "".join(data)
59
60  def tohex(data):
61    return "0x" + "".join([hex(ord(x)).replace("0x","").zfill(2) for x in data])
62
63  def fromhex(data):
64    out = ""
65    for i in xrange(0,len(data),2):
66      out += chr(int(data[i] + data[i+1],16))
67    return out
68
69
70  def main():
71    allow = set([fromhex(x.strip().replace("0x","")) for x in open('card_lists/↩
          allowed_cards','rU').read().split("\n") if x.strip() != ""])
72    print allow
73    pw = "lUBe43hOnEzin50IdeAl"
74
75    ser = setup_device()
76
77    if (ser is None):
78      return 0
79
80    while (1):
81      xor = 0
82      data = ser.read(10)
83
84      print tohex(data), "=>",
85      data = decrypt(pw,data)
86      rfid = data[1:6]
87      random = data[0] + data[6] + data[8] + data[9]
88      checksum = data[7]
89
90      print tohex(rfid),tohex(checksum),tohex(random), "=>",
91
92      for item in rfid:
93        xor ^= ord(item)
94
95      if (ord(checksum) != xor):
96        print "checksum Failure, Expecting", hex(xor) , "found", hex(ord(checksum↩
              ))
97      else:
98        if rfid in allow:
99          ser.write(encrypt(pw,random[0] + chr(1) + chr(0) + chr(0) + chr(0) + ↩
                chr(0) + random[1] + chr(1) + random[2] + random[3]))
100         print "granting access"
101       else:
102         ser.write(encrypt(pw,random[0] + chr(1) + chr(0) + chr(1) + chr(0) + ↩
                chr(0) + random[1] + chr(0) + random[2] + random[3]))
103         print "denying access"
104
105
106 try:
107   main()
108 except KeyboardInterrupt:
109   print "\nQuiting, Bye"
```

8.6.2 Testing Server

```
import serial
import time

def open_port():
  return serial.Serial('/dev/ttyUSB0',9600)

def enter_command_mode(ser):
  time.sleep(1)
  ser.write('+++')
  time.sleep(1)

  if(ser.read(3) != "OK\r"):
    return False
  return True

def at_command(cmd,ser):
  ser.write(cmd + "\r")
  if(ser.read(3) != "OK\r"):
    return False
  return True


def setup_device():
  at_commands = ["ATID4214","ATDL01","ATMY10","ATCN"]

  ser = open_port()
  if (enter_command_mode(ser)):

    for command in at_commands:

      if ( not at_command(command,ser) ):
        print "fail on:", command
        return None

    print "setup OK"
    return ser

  else:
    print "failed to enter command mode"
    return None

def encrypt(pw,data):
  data = list(data)
  data[0] = chr(ord(data[0]) ^ ord(pw[0]) ^ ord(pw[1]))
  for i in xrange(1,len(data)):
    data[i] = chr(ord(data[i]) ^ ord(data[i-1]) ^ ord(pw[i+1]))

  return "".join(data)

def decrypt(pw,data):
  data = list(data)

  prev_crypt = ord(pw[0])
  for (i,item) in enumerate(data):
    data[i] = chr( ord(data[i]) ^ ord(pw[i+1]) ^ prev_crypt)
    prev_crypt = ord(item)

```

```
58    return "".join(data)
59
60  def tohex(data):
61    return "0x" + "".join([hex(ord(x)).replace("0x","").zfill(2) for x in data])
62
63  def fromhex(data):
64    out = ""
65    for i in xrange(0,len(data),2):
66      out += chr(int(data[i] + data[i+1],16))
67    return out
68
69
70  def main():
71    allow = set([fromhex(x.strip().replace("0x","")) for x in open('card_lists/↩
          allowed_cards','rU').read().split("\n") if x.strip() != ""])
72    wrong_random = set([fromhex(x.strip().replace("0x","")) for x in open('↩
          card_lists/wrong_random_reply','rU').read().split("\n") if x.strip() != ""↩
          ])
73    wrong_checksum_in = set([fromhex(x.strip().replace("0x","")) for x in open('↩
          card_lists/wrong_checksum_received','rU').read().split("\n") if x.strip() ↩
          != ""])
74    wrong_checksum_out = set([fromhex(x.strip().replace("0x","")) for x in open('↩
          card_lists/wrong_checksum_reply','rU').read().split("\n") if x.strip() != ↩
          ""])
75    no_reply = set([fromhex(x.strip().replace("0x","")) for x in open('card_lists↩
          /no_reply','rU').read().split("\n") if x.strip() != ""])
76    incomplete_reply = set([fromhex(x.strip().replace("0x","")) for x in open('↩
          card_lists/incomplete_reply','rU').read().split("\n") if x.strip() != ""])
77    overflow_reply = set([fromhex(x.strip().replace("0x","")) for x in open('↩
          card_lists/overflow_reply','rU').read().split("\n") if x.strip() != ""])
78
79    print allow,
80    pw = "lUBe43hOnEzin50IdeAl"
81
82    ser = setup_device()
83
84    if (ser is None):
85      return 0
86
87    while (1):
88      xor = 0
89      data = ser.read(10)
90
91      print tohex(data), "=>",
92      data = decrypt(pw,data)
93      rfid = data[1:6]
94      random = data[0] + data[6] + data[8] + data[9]
95      if (rfid in wrong_checksum_in):
96        checksum = chr(ord(data[7]) + 1)
97      else:
98        checksum = data[7]
99
100     print tohex(rfid),tohex(checksum),tohex(random), "=>",
101
102     for item in rfid:
103       xor ^= ord(item)
104
105     if (ord(checksum) != xor):
106       print "checksum Failure, Expecting", hex(xor) , "found", hex(ord(checksum↩
            ))
107     else:
```

```
108         if rfid in wrong_random:
109           if (random[0] == "w"):
110             ser.write(encrypt(pw,"C" + chr(1) + chr(0) + chr(0) + chr(0) + chr(0)↩
                    + random[1] + chr(1) + random[2] + random[3]))
111           else:
112             ser.write(encrypt(pw,"W" + chr(1) + chr(0) + chr(0) + chr(0) + chr(0)↩
                    + random[1] + chr(1) + random[2] + random[3]))
113           print "sending incorrect random data"
114         elif rfid in wrong_checksum_out:
115           ser.write(encrypt(pw,random[0] + chr(1) + chr(0) + chr(0) + chr(0) + ↩
                chr(0) + random[1] + chr(0) + random[2] + random[3]))
116           print "sending wrong checksum"
117         elif rfid in no_reply:
118           print "not replying"
119         elif rfid in incomplete_reply:
120           ser.write(encrypt(pw,random[0] + chr(1) + chr(0) + chr(0) + chr(0) + ↩
                chr(0) + random[1] + chr(1) + random[2]))
121           print "sending incomplete reply"
122         elif rfid in allow:
123           ser.write(encrypt(pw,random[0] + chr(1) + chr(0) + chr(0) + chr(0) + ↩
                chr(0) + random[1] + chr(1) + random[2] + random[3]))
124           print "granting access"
125         elif rfid in overflow_reply:
126           ser.write(encrypt(pw,random[0] + chr(1) + chr(0) + chr(1) + chr(0) + ↩
                chr(0) + random[1] + chr(0) + random[2] + random[3] + random[0]))
127           print "sending deny access with extra data"
128         else:
129           ser.write(encrypt(pw,random[0] + chr(1) + chr(0) + chr(1) + chr(0) + ↩
                chr(0) + random[1] + chr(0) + random[2] + random[3] + random[0]))
130           print "denying access"
131 
132 
133 try:
134   main()
135 except KeyboardInterrupt:
136   print "\nQuiting, Bye"
```