

A Software Extension for Practical Attacks on RFID Devices

Hendrik Dettmer

September 25, 2007

Studienarbeit

Advised by: Dipl.-Ing. Timo Kasper



Department of Electrical Engineering and Information Sciences
Ruhr-University Bochum
Chair for Communication Security (COSY)
Prof. Dr.-Ing. Christof Paar

Contents

1	Introduction	1
1.1	RFID Overview	1
1.2	System Description	3
2	Communication	6
2.1	Reader to Card (Miller Code)	6
2.2	Card to Reader (Manchester Code)	11
3	Operating Modes	14
3.1	MENU Mode	14
3.2	Relay Mode	15
3.3	Reader mode	19
3.4	Card Mode	22
3.5	Output Mode	23
3.6	Reset Mode	24
3.7	Save Mode	25
4	The Visual C++ Program	27
4.1	Introduction	27
4.2	Read Data	28
4.3	Send Data	31
4.4	Operating Modes	32
4.4.1	Reader Mode	32
4.4.2	Card Mode	34
4.4.3	Information Mode	34
4.5	Some other functions	34
4.5.1	Decoder	34
4.5.2	CRC-Check	35
4.5.3	Save Mode	35
5	Future Prospects	36
6	Conclusion	37

Abstract

RFID (Radio Frequency Identification [1]) cards are used in our everyday world. We can pay with them and they are used for identification puposes.

This work will show how practical attacks with the RFID tool¹ can affect these cards. It enables convenient replaying and relaying the data of an RFID communication with the RFID tool. Especially the relay attack is very straightforward and it is very difficult to protect the protocol against it. During the attack, the communication can be recorded and hence analyzed after the attack. In this work functions were programmed, both for an Atmel AVR 8 bit microcontroller and for a PC, which allow for or simplify such attacks.

First a framework for the microcontroller was created. After that an interface was implemented which simplifies the device's handling in order that it can be used to search for security holes in RFID protocols and assist different types of side channel attacks.

¹device build by Timo Kasper [2]

1 Introduction

1.1 RFID Overview

RFID stands for Radio Frequency IDentification and is an automatic identification method. The RFID transponders can communicate contactlessly with a reader through the air and send their identification or other information over this channel. The idea of contactless identification devices was born in World War II [3] and developed to today's RFID chips which are as complex as a tiny computer. This chips occupy, with modern silicon wafer manufacturing, an area as small as $0.04 \times 0.04 \text{ mm}^2$, have a thickness of a sheet of paper [4] and a very low energy consumption.

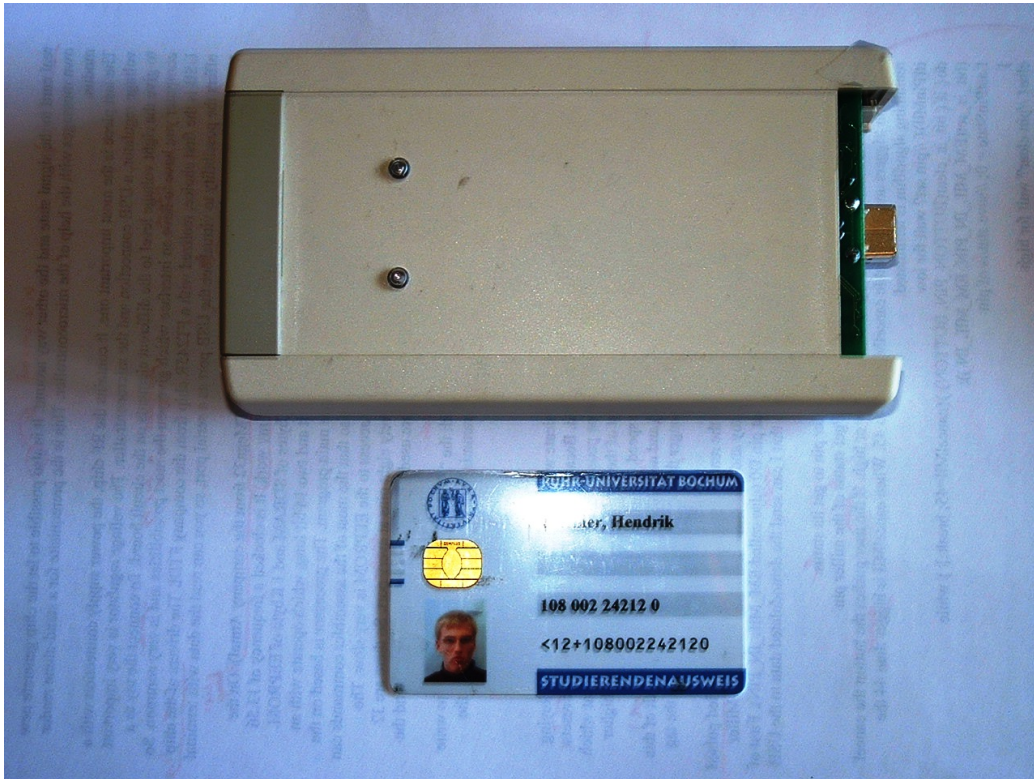


Figure 1: A Basic RFID System - Reader and Tag

An RFID system (which was in the focus of my work) contains two major

devices: the reader and the transponder or (in short) tag (see Figure 1). The reader can not only read the information from the tag, but it can also send data to the tag. Often the reader is connected to another device like a computer.

There are two large groups of tags. On the one hand the RFID tags which can communicate over a longer distance but have small complexity and computing power. On the other hand there are contactless smartcards like the new e-passport or a student identity card. These tags have a limited range (normally up to 10 cm) at which they can operate, as they demand for very high computing power. They can store many kByte of data and perform tasks like generating a pseudo random number. Only some of these chips have their own power source. Most of the contactless smartcards use the reader as a power source to operate; these chips are called passive tags. The reader provides energy for the chips, e.g. in form of a magnetic field. The antenna of the cards can capture this field via so called inductive coupling. To ensure a constant power source the tag uses a capacitor to bypass small interruptions of the field. The properties of proximity cards are defined in the ISO (International Organization for Standardization) Standard 14443A or 14443B [6] and communicate in the HF (High Frequency) range at 13.56 MHz. The most common type of cards, which is compatible with these standards, is the Philips Mifare card [8]. This paper concentrates on the ISO Standard 14443A. The standard defines amongst others the operating frequency, modulation codes, coding schemes and the anti-collision protocol.

Because of the nature of a wireless connection the fear has come up that attackers can spy out cards without notice of the owner. The attacker can read the card while it is in a pocket or a purse. The great advantage is that the transaction time is reduced. You only have to go through a detector and the reader reads the card in your wallet and reduces your balance. New inventions like wearable or implanted transponders in shoes and even under the skin of human beings are today used to gain access to restricted areas.

1.2 System Description

Timo Kasper's RFID tool is the basis for this work. For further information and an exact layout of the circuit board, please refer to [2]. Basically it can be divided into two parts. Number one is the fake tag which is the link to the genuine RFID-reader. It shall appear as a standard tag to the reader. Because of its own power connection (it is not dependent on energy from the reader like a genuine tag) it can detect the reader's field and demodulate the information contained in the field into a signal which is sent to the main RFID-tool. It can not only acquire this information of the reader, but it can also get a bitstream from the main tool and send it to the genuine reader employing a load modulation. The fake tag, as seen in Figure 2, has also some trimmable components to change the load modulation. So it is possible to tune it to the operating frequency.



Figure 2: **The Fake Tag**

The other part is the main RFID tool. A small antenna is connected with a coaxial cable to a RF (Radio Frequency) chip² which handles the communication with the genuine contactless smartcard. This chip can be accessed via microcontroller, but works after its initialization also without the microcontroller's help. This chip can detect the information contained in the field and relays this signals to the microcontroller. Also it is possible to let the chip generate new own messages with the help of the microcontroller. This is the main prerequisite for a stand-alone reader mode.

²EM Microelectronic EM4094 chip [16]

The main circuit board, Figure 3, is the most important part of the system. It contains the RF chip, the power supply connection with a voltage regulator, a FTDI (Future Device Technology International [13]) chip for USB (Universal Serial Bus) connection and the microcontroller. The voltage regulator is very important to provide the appropriate voltage level for the different elements on the circuit board. To connect the toy to a computer an interface is needed which provides a lot of possibilities and which is very common. So USB is the first choice, realized with an FT245R chip. The supplied Windows drivers offer the possibility to virtually see the USB port as a serial port, so the user can read data with terminal programs like HyperTerminal or Hterm [10].



Figure 3: **The Main RFID Tool**

The last element on the circuit board is the microcontroller, an Atmel AT-Mega32 [14]. The main focus of this work is programming this controller.

It is clocked at a frequency of 13.56 MHz and is equipped with 32 Kbyte of Flash RAM (Random Access Memory), 2 Kbyte of SDRAM (Synchronous Dynamic Random Access Memory) and 1 Kbyte of EEPROM (Electrically Erasable Programmable Read-Only Memory). The microcontroller has three timers, one with 8 bit and two with 16 bit which operate with an independent counter unit parallel to the main program. The processor is based on the RISC (Reduced Instruction Set Computer) architecture so that most assembler commands can be executed in one CPU (Central Processing Unit) cycle. A CPU cycle in this case is about 73.7 ns long. Note that the access to the EEPROM is very slow.

The microcontroller offers four ports (A-D) with eight pins at each port. So there are 32 possible ways to connect to the microcontroller. All of these possibilities are occupied in the current implementation. On the circuit board are also two buttons. One is on the left side of the layout and resets the whole board. The other one, just on top of the microcontroller, is a user programmable button which is used in the implementation. After initialization of the RF and FTDI chip the microcontroller enters an endless loop. This loop acts like an operating system. When the execution of a code segment is finished, the microcontroller does not shut down, but it executes the main loop.

For programming this controller I used the AVR Studio [14] with the add-on WinAVR³ to write and compile source code and AVRdude [7] and a programming adapter to copy this code into the microcontroller.

A further focus of my work is the development of an interface for the device. The aim is a GUI (Graphical User Interface), in which a user can see the output of the RFID toy and can easily communicate with it. To build this GUI I used Visual Studio C++ Express Edition by Microsoft [9] and the Windows SDK (Software Development Kit [15]).

³AVR-GCC [5]

2 Communication

2.1 Reader to Card (Miller Code)

The reader uses modified Miller code to communicate with a tag. This ASK (Amplitude Shift Keying) method operates with pauses in the electromagnetic field, see Figure 4. Because of the tag's internal capacitor these pauses do not de-energize the tag. Instead it can detect these pauses and analyze them. The standard rate at which a reader can send data to a tag is 106 kBit/s. Most of the tags and readers can operate at higher speeds. The first commands in a communication are always sent at a speed of 106 kBit/s, to provide compatibility.

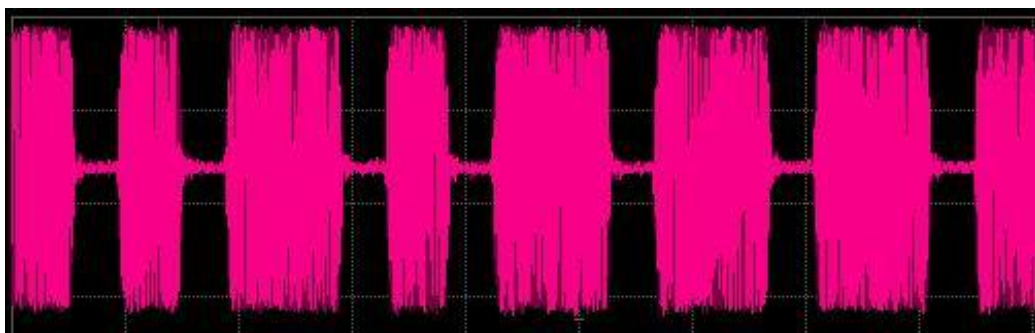


Figure 4: **Field with Miller Pauses**

The pauses can also be displayed as pulses (inverted), making it easier to see that the length of this pulses is between $2\text{-}3\,\mu\text{s}$. If the pauses would be too long, the tag's capacitor would drain and the tag would switch off.

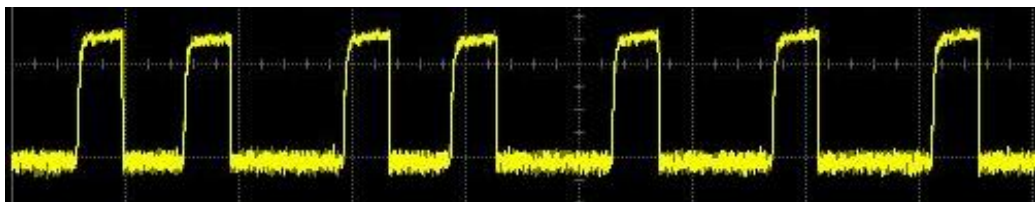


Figure 5: **Miller Code Displayed as Pulses**

The fake tag receives the electromagnetic field from the reader and translates it to modified pulsed Miller code, as described in Figure 5. To save bandwidth the signal is further converted to normal Miller code (Figure 6). This normal Miller code is sent to the RFID tool where the microcontroller demodulates this code into bits.

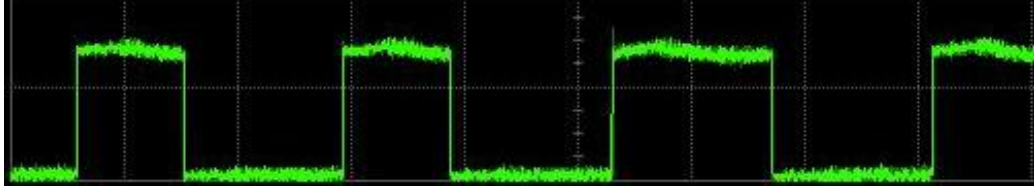


Figure 6: **Normal Miller Code**

The microcontroller receives this normal Miller code at pin D0 (also called EM_MIL_IN_PIN). First of all the code has to be translated into ones and zeros, so that the demodulated data can be sent to the USB port or saved in the EEPROM.

After the initialization of the variables the Miller pin is polled to get its state.

```
//get state of the Miller pin
PinMil = ( EM_MIL_IN_PIN >> EM_MIL_IN ) & 1;
```

It is now clear if the microcontroller has to wait for the next transition. When the user pushes the mentioned button the cancel condition triggers and a variable `cancelMil` is set to 45. When `cancelMil` is greater than 44 the algorithm for demodulating the Miller code is terminated. Note that the whole algorithm is embedded in a while loop which checks the cancel condition.

```
//pin set? wait for low
if(PinMil){
    do{
        if ( bit_is_clear(BUTTON_PIN, BUTTON) ) {
            cancelMil=45;
            break;
        }
    } while(1);
}
```

```

        }
    } while (bit_is_set(EM_MIL_IN_PIN, EM_MIL_IN ));

    //save state of pin
    LastPinState = 0;

}
//pin clear? wait for high
else{

    do{
        if ( bit_is_clear(BUTTON_PIN, BUTTON ) ) {
            cancelMil=45;
            break;
        }
    } while (bit_is_clear(EM_MIL_IN_PIN, EM_MIL_IN ));
    //save state of pin
    LastPinState = 1;
}

```

Now the algorithm enters this while loop to acquire a Miller code. In this loop the microcontroller waits actively for the next Miller transition.

```

do{
    //count cancelMil up
    cancelMil++;
    if ( cancelMil > 44 ) break;
} while (bit_is_set( EM_MIL_IN_PIN, EM_MIL_IN ));

```

If no pulse occurs after $26\mu\text{s}$ (cancelMil counted up to 45⁴) the waiting algorithm terminates. In a normal communication the maximum pause between two pulses is about $19\mu\text{s}$.

⁴Counting the variable up and checking it equals about 6 CPU cycles each time. So about 6 cycles times 45 increments times 73.7 ns equals about $26\mu\text{s}$

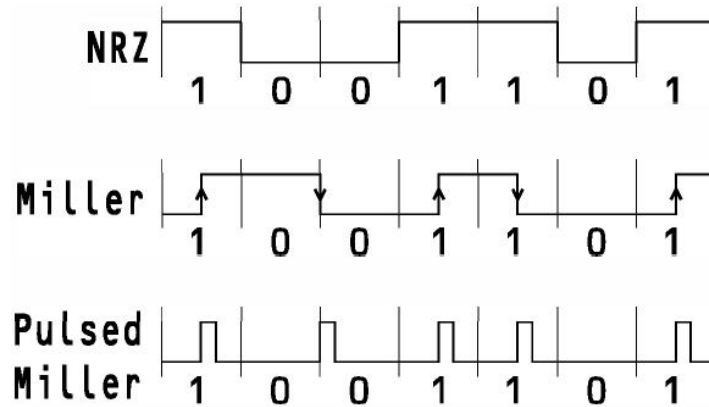


Figure 7: Miller Coding

A pulse in the middle of a bit period is a one and a pulse at the beginning of a bit period is a zero after a zero, compare with Figure 7. So there are limited possibilities for how long a pause between two pulses can be.

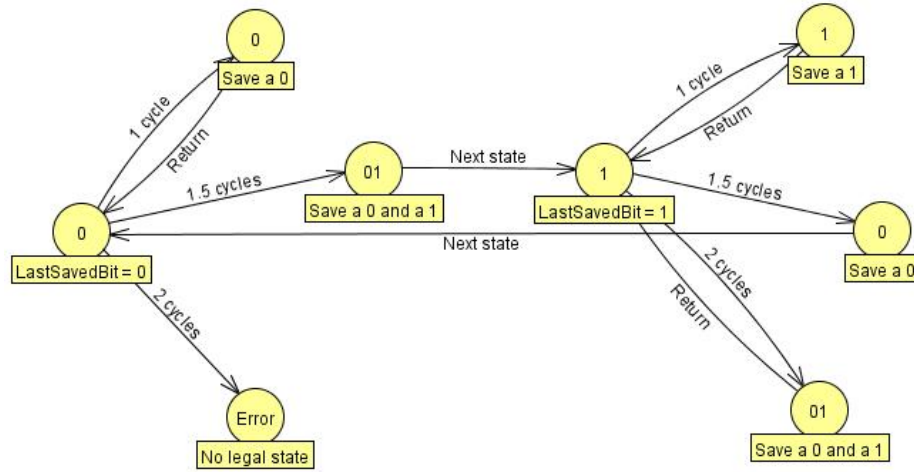
One problem I discovered was that the Miller code is not self-clocking. There is no direct connection between the clock cycle and the Miller pulses. So I had to find a way to distinguish between zeros and ones. For that I analyzed the pauses between two transitions and found a way to detect the different bits. There are only three different pause lengths between two transitions in a normal communication. The pause can be 1, 1.5 or 2 bit periods (clock cycles) long.

From this observation the timer value is saved and compared to three different periods⁵.

But I needed one last information which exactly defines which bits were sent: the last received bit before the pause, which then is stored in the variable `LastSavedBit`. The resulting state machine is depicted in Figure 8.

The obtained bits are now arranged to bytes to minimize the memory needed. Every bit is left-shifted into a byte. Each time one bit is saved in the array, the counter `BitCounter` is increased and the algorithm checks if the current byte is full. Each byte is then saved in an array. Then the microcontroller actively waits again for the next transition.

⁵9.44 μs (1 bit period), 14.16 μs (1.5 bit period) and 18.88 μs (2 bit period)

Figure 8: **State Machine**

If no new transition occurs the value of `cancelMil` reaches 45 and the while loop terminates. Now the button is checked and the size of this array is written into the second array segment.

2.2 Card to Reader (Manchester Code)

To send its data, the card employs load modulation of the electromagnetic field. It modulates the field by using the Manchester code. It works like a transformer in which the current is changed. The other side can detect these changes and capture information. The maximum data transfer rate is again 106 bit/s.

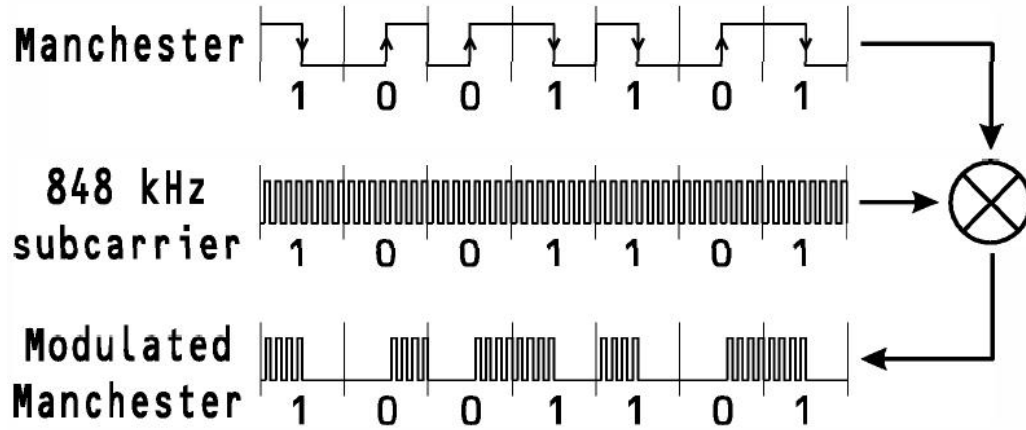


Figure 9: Manchester Coding

The load modulation signal sent through the channel is derived from the bitstream as illustrated in Figure 9. The tag transforms the field in this way and the RF chip can demodulate the signal. This signal is received by the microcontroller on pin D3 (also called EM_MAN_OUT_PIN). It is easier to decode compared to the Miller code because of its self-clocking attribute. First of all some variables are initialized and the program waits for the first bit to appear on the pin. This first bit is not really captured it only starts the algorithm. To obtain the bit ⁶ it is always pre-stored at the beginning of the array.

```
do{
    if ( bit_is_clear( BUTTON_PIN, BUTTON ) ){
        cancelMan=15;
        break;
    }
}
```

⁶The first bit is also called SOC (Start of Communication) and is always one in the Manchester code.

```

    }
} while bit_is_clear(EM_MAN_OUT_PIN, EM_MAN_OUT ));
// 1. half bit (SOC) wait for high

do{
    if ( bit_is_clear( BUTTON_PIN, BUTTON ) ){
        cancelMan=15;
        break;
    }
} while (bit_is_set(EM_MAN_OUT_PIN, EM_MAN_OUT ));
// 1. half bit (SOC) wait for low

```

The button is used again to let the user cancel the algorithm. If the user aborts the program the variable `cancelMan` is set to 15. Next thing is setting `timer0` prescaler to 8 and starting the timer. As long as the cancel condition is not fulfilled, the following program text is performed.

```

if(PinBefore){
    do{ }
    while(TCNT0 < 9); // 5,3 s*10-6
} // wait till clock begin

else{
    do{}
    while(TCNT0 < 12); // 7 s*10-6
}
//save pin level
PinBefore = (EM_MAN_OUT_PIN >> EM_MAN_OUT) & 1;

```

This active waiting is used to compensate the different lengths of a high and a low pulse which occurred in practice, and to skip more than a half bit period. If the last transferred bit was a one the high time is only about $5\mu\text{s}$ long. Whereas if the last transferred bit was a zero the pulse is about $7\mu\text{s}$ long. After skipping the first half of the bit period, the state of the pin in the second half of that bit period is saved to the variable `PinBefore`.

```

if ( PinBefore )
    do{ cancelMan++;

```

```
        if ( cancelMan > 11 ) break;
    }
    while (bit_is_set( EM_MAN_OUT_PIN, EM_MAN_OUT ));
else
    do{ cancelMan++;
        if ( cancelMan > 11 ) break;
    }
    while (bit_is_clear( EM_MAN_OUT_PIN, EM_MAN_OUT ));
TCNT0 = 0;
```

Now the program waits for the end of that bit period. If there is no transition in the level of the pin for a long time (10-12 μ s) the algorithm for demodulating the Manchester code is aborted. In case of a successful acquisition the timer is set to zero. Then the received bit is saved in an array. These operations are looped until a cancel condition is met. So, basically, the program skips the first half of a bit period and then only takes the value of the pin. Now the program waits for the change in level at the pin to identify the bit period's end. Simultaneously it synchronizes with the received Manchester code. The timer is set before saving and is compared in the first if-else-condition after the saving is done. So the required time for storing a bit is not important.

3 Operating Modes

There are 5 modes, for different purposes, in which the microcontroller can operate. They developed over the duration of the work. For a change of mode the user quits the current mode and restarts the device. Then the user can enter a letter and send it to the RFID tool. For this a USB connection must be established. The operation mode in which a user can access this menu is called "MENU" (this is displayed via USB on the computer screen).

3.1 MENU Mode

First of all the string, containing the text "MENU Mode", is sent to the USB port to be displayed on the screen. Now the while loop is entered and the algorithm is waiting for an input by the user.

```
//display current mode
FT_Send(convertSend,11);

while(rcv_length==0){ //wait for data
    rcv_length=FT_Receive(USBRCV);
}
```

When the function `FT_Receive` gets data from the USB connection it stores the data in the array `USBRCV`. The size of the received array is stored in the variable `rcv_length`. So this integer no longer equals zero and the while loop is canceled.

```
//select first letter from USBRCV
//to select the new mode
switch(USBRCV[0])
```

The first array element is the letter which specifies the next mode. So with this switch assignment a menu element is selected. A second letter may follow for certain modes, specifying special options.

The available menu elements are:

- 'r' : relay mode (the communication between card and reader is relayed and captured)
- 'l' : reader mode (the RFID tool functions as a reader)
- 'c' : card mode (the RFID tool functions as a card)
- 'k' : reset mode (all memory pointers are reset)
- 'o' : output mode (data from the EEPROM is sent to the PC)

3.2 Relay Mode

The relay mode is split into two parts of code. In one small code piece the mode is selected and in another very large piece of code the procedure is performed. The MENU mode contains the small piece of code, by which the microcontroller is put into the relay mode.

```
case 'r':                //relay mode

    //write mode to EEPROM
    eeprom_write_byte( ee_state1, USBRCV[0] );

    //write mode to variable
    state = USBRCV[0];

    //write second mode to EEPROM
    eeprom_write_byte( ee_state2, USBRCV[1] );

    break;
```

If an 'r' is sent over the USB connection, the microcontroller is set to relay mode. To save this setting in case of a restart it is also written to the EEPROM. After a restart or a loss of energy the microcontroller reads out the letter from the EEPROM and puts itself in the correct mode.

The user can decide whether the collected data is saved to the EEPROM or is directly sent to the computer. This decision is contained in the second

letter. So when the user sends 'ru' this means that the relay mode 'r' should transfer all collected data to the USB port 'u'. The other possibility is 're' for the EEPROM.

The selected mode is again displayed to the user. All the required variables are initialized and the correct registers for the pins are set, i.e. to input, output or tristate. The port register defines the pull-up state of the pins. The DDR register defines if the pin is an input or an output pin. Now the second letter is collected from the EEPROM, to know from which device the input is coming.

```
//read the second letter
eeSave = eeprom_read_word( (uint16_t*)ee_relaPL );

if(eeprom_read_byte( ee_state2) == 'u') {
    usb = 1;                //if data from USB
}
else{
    usb = 0;                //else data from EEPROM
}
```

The microcontroller then enters an (infinite) while loop which can only be canceled by the user pushing the button or resetting the device. The device is now ready to relay the RFID communication.

The first step of a communication is a request by the reader, so a Miller code must be sent through the channel. This communication is captured as explained in Chapter 2.1. After all data from the reader is stored in the array `dataMil` the variable `usb` is checked to know where to send the captured communication.

```
if(usb){
    //send data to USB
    FT_Send( dataMil, index+1 );
}
else{
    //size of the tmp array
    jCount = rArrayCoun;
    endForLoop = rArrayCoun+index+1;
```

```

    for(jCount = rArrayCoun; jCount<=endForLoop; jCount++){
        //store data in tmp array
        relayArray[jCount] = dataMil[jCount-rArrayCoun];
    }
    //refresh to new size
    rArrayCoun = rArrayCoun+index+1;
}

```

If the microcontroller is directly connected to the USB port the complete array is sent to the FTDI USB chip with the `FT_Send` function. But if the data is supposed to be stored in the EEPROM, it is copied to a temporary array `relayArray`. First I tried to save the data directly to the EEPROM but the access was too slow. It took too much time so that further communication could not be captured. So the temporary array is stored to the EEPROM after the whole communication is relayed. Now that the first reader-card-communication is captured and saved, the card-reader-communication (Manchester code) must be recorded. This communication is captured and saved using the method described in Chapter 2.2.

The card-reader and reader-card communication capturing process is executed as long as new data comes over the channel. If no new data is sent the while loop waits for the user to push the button. After the button is pushed the collected data is saved.

First of all it is checked if the data must be sent to the USB port or must be stored in the EEPROM.

```

if(!usb){
    //too much data to save in EEPROM
    //warn user
    if( eeSave + rArrayCoun > 1023 ){
        clrBit(GREEN_PORT, GREEN);           //Low=On
        setBit(GREEN_PORT, GREEN);           //off
        clrBit(GREEN_PORT, GREEN);           //Low=On
        setBit(GREEN_PORT, GREEN);           //off
        clrBit(GREEN_PORT, GREEN);           //Low=On
        setBit(GREEN_PORT, GREEN);           //off
        break;
    }
}

```

```

    }
    //save data
    eeprom_write_block( relayArray,(uint16_t*)eeSave, rArrayCoun);
    //increase pointer
    eeSave = eeSave + rArrayCoun;
    //save pointer
    eeprom_write_word( (uint16_t*)ee_relaPL, eeSave);
}

```

If the data is supposed to be stored in the EEPROM, it is checked if the array is larger than the available EEPROM size. To check this, the start address of the saved array `eeSave` is added to the size of the array `rArrayCoun` and is compared to the size of the EEPROM (1024 Bytes). If the array is too large the green led blinks to warn the user. The array is saved in the EEPROM with the instruction `eeprom_write_block`. After this the memory pointer is increased and is also saved in the EEPROM, indicating the next free position to write to. For the other case, in which the data sent to the USB port is considered after each demodulation, the function `FT_Send` is called.

```

clrBit(RED_PORT,RED);                //Low=On
countertmp = 0;
for(countertmp = 0; countertmp < 51; countertmp++){
    //red led on
    _delay_ms(100);
}

setBit(RED_PORT,RED);                //OFF
countertmp = 0;
for(countertmp = 0; countertmp < 51; countertmp++){
    //red led off
    _delay_ms(100);
}

clrBit(RED_PORT,RED);                //Low=On

//user pushed button
if(bit_is_clear( PINB,2 )){
    //override state=mode in EEPROM

```

```

        eeprom_write_byte( ee_state1, '1');
        //override state=mode in variable
        state = 1;
        setBit(RED_PORT,RED);           //OFF
        clrBit(GREEN_PORT,GREEN);      //green led on
    }

```

During these for-loops, which are executed when the user has pushed the button, the red led blinks. This allows the user to decide whether to quit the relay mode or to restart it. If the user pushes the button long enough the relay mode quits. In this case a '1' is stored in the EEPROM and in the variable which contains the letter of the current mode. So the microcontroller enters the menu mode and the user can choose which mode is going to be executed next. If the user releases the button while the red led is flashing, the array is stored in the EEPROM. The microcontroller is still in the relay mode and can collect more data.

3.3 Reader mode

In the reader mode the device functions like a genuine RFID reader. When this mode is chosen in the menu, the letter 'r' is stored in the state variable and in the EEPROM memory location `ee_state1`. Now the current mode is displayed. Again the correct input and output pins are set. A reader sends commands to a tag with a Miller code, so the `EM_DIN` pin must be set as an output. Like in the relay mode the user can cancel the operation with a push on the button. As long as the button is not pushed the following code is executed.

```

//REQA is send
if(milIndex == 0){

```

⁷Request Command Type A

```

        //larger delay
        _delay_ms(100);
        //EEPROM address is set to start
        eeAddy = 0x10;
    }
    else{        //other command is sent
        //only little delay
        _delay_us(90);
    }

```

In the next step it is checked, if the data which is going to be sent to the tag comes from the EEPROM or the USB port. There is a special format for the commands to be transferred to the microcontroller. The letter 'I' must be in the first array position to indicate that the following command is a Miller code. The second byte to be stored is the size of the array. So the program knows how many bytes it has to read from the EEPROM or the USB port. The next bytes contain the payload which is to be sent.

```

//data from EEPROM
if(!usb){
    //get size of the command to send
    eeSize = eeprom_read_byte( eeAddy );
    if(eeSize == 0){
        //if size equals zero cancel the mode
        cancelOp = 17;
    }
    else if(eeSize == 1){
        //if size equals one - a short frame is sent
        eeAddy++;
        eeShort = eeprom_read_byte( eeAddy );
        //format short frame
        EM_SendShortFrame(eeShort);
    }
    else{
        eeAddy++;
        //read command from EEPROM
        eeprom_read_block( eeStd, eeAddy, eeSize );
        //format standard frame

```

```

        EM_SendStandardFrame(eeStd, eeSize);
    }
    //calculate new address for the next command
    eeAddy = eeAddy + eeSize;
}
//data from USB
else{
    do{
        //get command from USB port
        rcv_length=FT_Receive(USBRCV);

        //as long as there are no new commands
        //or the button has been pushed
    }while(USBRCV[0] != 'I' && bit_is_set(PINB, 2) );

    //second array element is the size
    ftSize = USBRCV[1];
    if(ftSize == 0){
        //if size equals zero cancel the mode
        cancelOp =17;
    }
    else if(ftSize == 1){
        //if size equals one a short frame is send
        eeShort = USBRCV[2];
        //format short frame
        EM_SendShortFrame(eeShort);
    }
    else{
        uint8_t ftInd = 0;
        for(ftInd=0; ftInd<ftSize; ftInd++){
            //read command from USB
            eeStd[ftInd] = USBRCV[2+ftInd];
        }
        //format standard frame
        EM_SendStandardFrame(eeStd, ftSize);
    }
}
//modulate the array with Miller code

```

```
ML_tmp = EM_DoTheMiller();  
//send array to tag  
EM_SendMiller(ML_tmp);
```

After the command is sent to a tag the microcontroller waits for its answer. Therefore the Manchester demodulation code from Chapter 2.2 is executed. If the tag's answer is successfully captured the variable `milIndex` is increased by one and the next command is sent to the tag. On the other hand if no answer is captured the variable is set to zero and the next command which is going to be sent is again a `REQA`. The answer of the tag is again stored in the EEPROM or is directly sent to the USB port.

3.4 Card Mode

In this mode the device pretends to be an RFID tag (RFID card). Like the relay or the reader mode this mode is also be selected in the menu mode, but with the letter 'c'. The current mode is displayed when a USB connection is detected and then the device waits for data. In a normal RFID communication the reader must initialize the data transfer, so now the card must wait for the `REQA`⁸ to be sent. To receive this command the demodulation code from Chapter 2.1 for Miller code is executed. If valid data is intercepted, this data is redirected to the USB port or saved in the EEPROM. After intercepting the communication `timer0` is set to zero to measure the time between the last intercepted bit and the point in time at which to reply. This time is the required FDT (Frame Delay Time) between request and answer. When the card does not wait this time before it sends the answer, the reader does not accept the bits. When the timer is started the intercepted bytes are saved and sent with the same program code as in the reader mode, compare with Chapter 3.3. Now the answer is fetched, which has to be formatted like in the reader mode, except that the first byte of an array begins with an 'A' instead of an 'I' (for M'A'nchester instead of M'I'ller). The beginning of the array in the EEPROM is stored in the variable `ee_valc01`. After the answer is formatted with the instruction `EM_SendStandardFrame` and `EM_DoTheMan` (instead of `EM_DoTheMiller`) `timer0` is polled and when the frame delay time is reached the information is sent to the reader.

⁸Request Command Type A

```

//modulate the array with Miller code
ML_tmp = EM_DoTheMan();
//wait until the FDT is passed
while(TCNT0%140!=0){}
//send array to reader
EM_SendMan(ML_tmp);

```

Now the program waits for the next request by the reader or an abort by the user. If the user aborts the routine the mode is restarted or the user can get back to the menu mode, as described in Chapter 3.3

3.5 Output Mode

This mode copies the saved data from the EEPROM to the USB port. It is selected with the letter 'o'. The starting memory addresses for the 3 different arrays are well known (they are defined in the library `eeeprom.h`). To know how much data must be retrieved from the memory the pointers are loaded. Every time something has been saved in the EEPROM the pointer of the last saved byte has also been increased and saved to the memory.

```

//load relay mode pointer
relayPointer = eeprom_read_word((uint16_t*)ee_relaPL);
//load reader mode pointer
readerPointer = eeprom_read_word((uint16_t*)ee_reaWPL);
//load card mode pointer
cardPointer = eeprom_read_word((uint16_t*)ee_carWPL);

```

After the pointers are loaded the program checks if any data has been saved by the different modes. In the relay mode, for example, the starting memory address is 256. When the relay mode has saved something in the EEPROM the pointer was incremented and this value is bigger than `relayP` (256). Through subtracting `relayP` (256) from the pointer value the size of the saved array is calculated. Now an array of this size is read from the EEPROM.

```

//something saved?
if( relayPointer > relayP){
    //size of the saved array
    tmp = relayPointer-relayP;

```

```

    //read the array
    eeprom_read_block(myByteBuffer, relayP, tmp);
}

```

The command `FT_Send`, which sends the data to the USB port, can only send arrays which are no larger than 256 elements. To make sure that the array is not larger than these 256 elements, the loaded array is formatted into smaller fractions. As long as the size of the array is larger than 256 subtract 254 from this size and copy the next 254 elements from this array to a temporary array. After the elements are moved, this smaller array can be sent with the `FT_Send` command.

```

//as long as there are more elements
while(tmp > 256){
    tmp = tmp - 254;
    for(counBuf = 0; counBuf<=254; counBuf++){
        //move the elements into the temporary array
        sendBuffer[counBuf]=myByteBuffer[tmp + counBuf];
    }
    //send this array
    FT_Send(sendBuffer,254);
}

```

Finally get the last left elements from the array and send it to the PC.

```

tmp2 = tmp;
for(counBuf = 0; counBuf<=tmp2; counBuf++){
    //move the last elements into the temporary array
    sendBuffer[counBuf]=myByteBuffer[counBuf];
}
//send this array
FT_Send(sendBuffer,tmp2);

```

3.6 Reset Mode

The reset mode, selected with the letter 'k', sets the pointers to the EEPROM to their initial values, after the memory has been read. To be able to read the data from the memory more than once, the reset mode was split off the

output mode. Now the device can be connected to two different computers and both can read the captured data from the EEPROM. Note that, after executing the reset mode, all previously acquired data is lost.

```
//set to initial value 256
eeprom_write_word((uint16_t*)ee_relaPL, relayP);
//set to initial value 512
eeprom_write_word((uint16_t*)ee_reaWPL, readerWrP);
//set to initial value 768
eeprom_write_word((uint16_t*)ee_carWPL, cardWrP);
```

Now the device is ready to capture new data.

3.7 Save Mode

With the save mode the user can transfer new data into the EEPROM for the reader or card mode. The data in the EEROM is overwritten and no more than 64 bytes can be saved. To select this mode the letter 's' must be sent. The second letter determines in which area the data is stored ('r' for reader, 'c' for card). After this second letter the size must be specified and sent as a number at the third position.

The first step is reading out the size of the array to be stored, and then copying this array to a temporary array.

```
//get size of array
uint8_t length_ee = USBRCV[2];
uint8_t i = 0;
for(i=0; i <= length_ee; i++){
    //copy received array into temporary array
    myByteBuffer[i] = USBRCV[i+4];
}
```

Now the program checks in which area the data is to be stored. The last step is saving it in the EEPROM.

```
//data for reader mode
```

```
if(USBRCV[1] == 'r'){
    eeprom_write_block( myByteBuffer, 0x10, length_ee);
}
//data for card mode
else if(USBRCV[1] == 'c'){
    eeprom_write_block( myByteBuffer, 0x50, length_ee);
}
```

4 The Visual C++ Program

4.1 Introduction

To access the microcontroller in the first place the Windows HyperTerminal[12] was used. To get a better presentation of the data and change the modes more easily, a Visual C++ program was coded. With this program the user can select the mode in which the device operates and can see exactly which bytes the device sends back. Other functions allow him to save the displayed data in a file or let him execute reader and card USB modes. The USB reader and card modes are accessed by buttons.

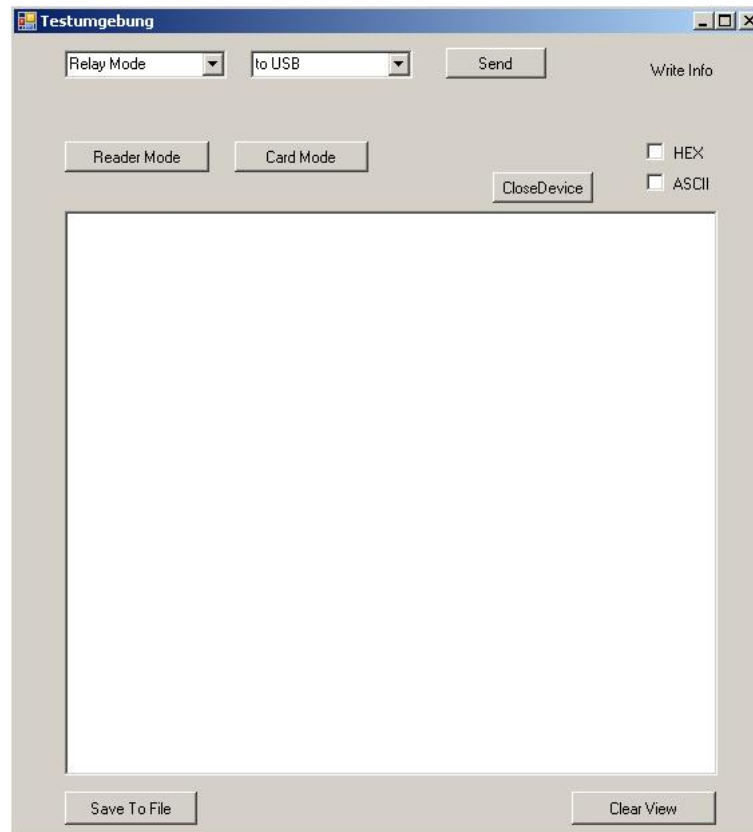


Figure 10: **The Visual C++ Program**

As described in Figure 10 the upper part of the window are two drop down

menus in which the user can select the mode for the device and where the captured data will be stored. The button **Send** performs the selected action. The small label on the right gives the user information about the state of the **write**-function (e.g. if the data has been written to the device). If the device is in the reader or card mode and the data will come from the USB connection, the user can start the reader or card mode of the C++ program by clicking one of the two buttons. The two buttons at the bottom of the window let the user save the whole text from the text box in a file or clear the complete text box.

4.2 Read Data

First I coded the calculation and presentation of the received data. This is the main job of the program and performed by the operation **readingFT** which is described next. After some variables are initialized the connection to the device is opened by the command **deviceOpen**. This command first fetches the **ftHandle** and loads it to its own address-space. The **ftHandle** specifies the FTDI device at the USB port. Each device gets its own number (the first one '1', the second one '2' and so on). If the handle is null no connection has been established so the command **FT_Open** is executed. These FT commands are defined in the file **ftdi.h** and are implemented in **FTD2XX.dll**, which are both part of the driver package supplied by the manufacturer.

```
private: System::Void deviceOpen(){
    //open connection to device
    FT_STATUS ftStatus;
    FT_HANDLE ftHandle1 = ftHandle;
    if(ftHandle == NULL){
        //open connection, save connect number in ftHandle
        ftStatus = FT_Open( 0, &ftHandle1);
        if(ftStatus == FT_OK){
            richTextBox1->AppendText("Device Open");
        }
        //set Event hEvent when data is coming
        ftStatus = FT_SetEventNotification(ftHandle1, FT_EVENT_RXCHAR, hEvent);
    }
    ftHandle = ftHandle1;
}
```

The number of the current connection is saved in the handle and the return status code `ftStatus` is checked. To trigger an event when new data is ready to collect, the command `FT_SetEventNotification` is executed. In this function the event `hEvent` is linked to the device with the handle `ftHandle`. And `hEvent` is set when this device has sent a character. After this execution the handle is copied back into the global `ftHandle`.

To use events in this code, the function `readingFT` must be started in a thread. To do this the operation `startThread` is called and the event `hEvent` is created at the beginning of the program. In `startThread` a new thread is created and started.

Now the connection to the device is open and data can be received. The function `readingFT` waits for the `hEvent` to be set. The parameter `INFINITE` in the call of `WaitForSingleObject` means that this thread waits for an infinitely long time to get triggered.

```
//wait for hEvent --> data is available
WaitForSingleObject(hEvent, INFINITE);
```

If the event is set the next command is executed. `FT_GetQueueStatus` gets the number of bytes which will be displayed, and saves this number in `RxBytes`.

Now each single byte is collected from the FTDI buffer. The function `FT_Read` gets one (therefore the '1' in the call) byte from the device with the handle `ftHandle` and stores it in the variable `RxByte`. `BytesReceived` contains the number of bytes read by this function (in this case one).

```
//read one byte from device
ftStatus = FT_Read(ftHandle, &RxByte, 1, &BytesReceived);
```

In the next steps the received byte is formatted to gain more possibilities for displaying data. So the first 4 bits and the last 4 bits of the byte are copied into extra bytes to permit a hexadecimal presentation. But before the byte is displayed the program checks in which mode (reader, card, information) the program is in. Depending on the selected mode, the byte is

processed in a different manner. If no mode is selected the byte is tested if it is an 'A'(reader), 'I'(card) or 'N'(information). Should the byte be one of these letters the associated mode is selected and displayed with the operation `richTextBox1->AppendText`.

```
switch(RxByte){
    case 65:                                     //'A'
        richTextBox1->AppendText("Manchester:");
        richTextBox1->AppendText(Environment::NewLine);
        //if RxByte is 'A' set Boolean true
        bManchester = true;
        //set index to one -- first byte
        index = 1;
        break;
    case 73:                                     //'I'
        richTextBox1->AppendText("Miller:");
        richTextBox1->AppendText(Environment::NewLine);
        //if RxByte is 'I' set Boolean true
        bMiller = true;
        //set index to one -- first byte
        index = 1;
        break;
    case 78:                                     //'N'
        richTextBox1->AppendText("Info:");
        richTextBox1->AppendText(Environment::NewLine);
        //if RxByte is 'N' set Boolean true
        bInfo = true;
        //set index to one -- first byte
        index = 1;
        break;
    default:                                     //no genuine data is coming
        //set all to false
        bManchester = false;
        bMiller = false;
        bInfo = false;
        break;
}
```

If no mode is set and the received byte does not match one of these letters, all `boolean` values are set to false. After these checks each received byte is written into `richTextBox1`. The output format is determined by the two check boxes. If the first one is set, the corresponding ASCII (American Standard Code for Information Interchange) character is displayed, and if the second is set the byte is displayed in the hexadecimal representation.

```
//radio button ascii is checked
if(checkBox1->Checked){
    richTextBox1->AppendText(asciiString);
}
//radio button hex is checked
if(checkBox2->Checked){
    richTextBox1->AppendText("->" + hexString + " ");
}
```

After the data is displayed the thread is going to sleep as long as no new data is received.

4.3 Send Data

For sending data to the device, the function `sendingFT` was implemented. It obtains a `char` array and the size of this array as inputs. Then `label1` is set to "Write Begin" to signalize to the user that the program sends data. After that the receive USB buffer of the device is purged with the command `FT_Purge`. Now with the operation `FT_Write` the array `tmpSource` is sent to the device specified in `ftHandle`. The parameter `length` is the size of the array and `ret` is a variable for the number of bytes received by the device. It is not necessary to check `ret` because if the algorithm does not succeed in sending the complete array, the variable `ftStatus` is not set to `FT_OK`.

```
label1->Text = "Write Begin";
//clear receive buffer of device
ftStatus = FT_Purge(ftHandle, FT_PURGE_RX);
//send array tmpSource to device
ftStatus = FT_Write(ftHandle, tmpSource, length, &ret);
//everything ok
if(ftStatus == FT_OK){
```

```
        label1->Text = "Write complete";
    }
    else{
        //there was a problem
        MessageBox::Show("Write Problem");
        return;
    }
```

Accordingly, it is possible to send data to the device by calling the function `sendingFT(char tmpSource[], int length)`.

4.4 Operating Modes

4.4.1 Reader Mode

If the reader mode in the device is selected and the data to send comes from the USB connection, the correct data must be sent quickly after the last data is received from the card. To react each time new data is received, a new thread was implemented, the reader mode thread. After the user sets the device to reader mode and clicks the button reader mode in the GUI, this thread is created and started. Then the function `readerMode` is executed. This function contains a counter which counts the commands which have been sent to the card. The first one is a REQA. An array is loaded with the correct bytes and then the function `sendingFT`, with these bytes as an argument, is called. Now the reader mode thread is going to sleep and waits for the answer of the tag. The RFID tool collects the answer of the tag and sends it back to the computer. The first byte in this answer from the microcontroller is an 'A' so the operation `readingFT` triggers and saves the arriving data for the reader mode thread.

Now each single byte is processed separately. So the size of the array, the number of information bits in the last byte and the rest of the data is copied in local variables. After the last byte is received the array is formatted with the function `decoder`, see Section 4.5.1. After calling the `decoder` function the array is formatted for presentation and can be displayed in `richTextBox1`. After the last byte is displayed the boolean `bManchester` is set to false and the event `sEvent` is triggered to wake up the reader mode thread.

```

if(index==1){
    //get number of bytes
    manIndex = RxByte;
    index++;
}
else if(index==2){
    //get number of additional bits
    manBitCou = RxByte;
    index++;
}
else if(index==manIndex){
    //last byte received
    //put additional bits left
    manBytes[manIndex-3] = (RxByte << (8-manBitCou));
    //calc number of bytes
    manCounter = (8 * (manIndex - 3) + manBitCou) / 9;
    if(manCounter == 0)manCounter = 1;
    if(manBitCou == 0)manIndex++;
    //send bytes to decoder
    manOutBytes = decoder(manBytes, manCounter);
    //other functions can use the array
    globalArray = manOutBytes;
    //and the number of bytes
    globalNumber = manCounter;
    for(int i=0; i<manCounter; i++){
        //hex encoding
        Byte m1 = hexDigits[manOutBytes[i] >> 4];
        Byte m2 = hexDigits[manOutBytes[i] & 0xF];
        charbuf2[0] = m1;
        charbuf2[1] = m2;
        //ascii encoding of hex numbers
        String^ manchester = ascii1->GetString(charbuf2);
        //output
        richTextBox1->AppendText(Environment::NewLine);
        richTextBox1->AppendText("0x"+manchester);
    }
    richTextBox1->AppendText(Environment::NewLine);
    richTextBox1->AppendText("Manchester End");
}

```

```
        //set bool to false no more Manchester code
        bManchester = false;
        //set event sEvent for reader mode
        SetEvent(sEvent);
    }
    else{
        //get byte, put it into a array
        manBytes[index-3] = RxByte; index++;
    }
}
```

4.4.2 Card Mode

This mode works just like the reader mode, see Chapter 4.4.1. A new thread is created which sleeps until new data for analyzing comes from the device. One difference between those two modes is that the card mode must first wait for the REQA of the genuine reader and then send data to the device. Like the reader mode it calculates and checks the CRC (Cyclic Redundancy Checking) with the function `crcCheck` and lets the function `decoder` (see Chapter 4.5.1) determine if the parity bits are correct. After the request is checked the function `cardMode` sends the next command to the microcontroller using the function `sendingFT`. Now the thread sleeps again.

4.4.3 Information Mode

This mode is just for the status messages of the device. If an array that starts with an 'N' is captured by the program, the next array elements are interpreted as ASCII letters. They will be displayed directly in the text box.

4.5 Some other functions

4.5.1 Decoder

This function gets an array and the size of this array as input parameters. First it eliminates the parity bits in this array. It catches every ninth bit and checks if the eight leading bits are an odd number of ones. If the result equals the parity bit, this bit is deleted. After all parity bits are deleted the array with the payload is returned.

4.5.2 CRC-Check

This function gets an array of bytes as input and can calculate the CRC-16-CCIT [11] for these bytes or checks if the last two bytes are the correct checksum for this array. To calculate, the last argument must be a 1 and to check the data this `integer` must be a 0. The middle number (`numbBytes`) must be equal to the number of payload bytes. This number has to be the amount of bytes without the checksum bytes if the user wants to check an array.

4.5.3 Save Mode

This mode in the drop down menu is special because when it is selected a new window opens, in which the user can write new data. These hexadecimal numbers are converted and then sent to the device to save them to the EEPROM.

5 Future Prospects

The C++ program can provide other protocols. At this stage the program can communicate with cards using the ISO 14443-3 protocol. To analyze and understand the complete communication between a reader and a tag, the appropriate protocol, which normally extends the ISO protocol, must be implemented. After the implementation the protocol associate commands and data can be traced.

With this knowledge an active man-in-the-middle attack could send other data (previously stored) instead of the correct answer. To alter the genuine data in real-time and to try to actively attack the protocol is a lot harder. The microcontroller is not capable of altering encrypted data, but with a connection to a computer the data could be decrypted, altered and encrypted (it depends on the employed cryptography).

The ISO protocol contains many delay times. In the practice, especially the RFID tag did not abide these conventions. Maybe it is possible to delay data to the card long enough for a decryption and a new encryption.

For a DFA (Differential Fault Analysis) the assembler or the C++ program could change some bits in the transferring data. The result is a fault in the decryption of that data which could point out some bits of the key used.

6 Conclusion

An assembler and a C++ program were coded for use with the RFID tool. These two programs extend the possibilities of this device. Through the direct connection between a computer and the device, also on the software-layer, the computing power of a normal computer can be used for attacks. To enable the device for these new tasks the demodulation of the Miller and Manchester code had to work fine. Especially the coding of the Miller demodulation was difficult because of its not-self-clocking property. The device now functions as a card or reader and can save the captured data in the EEPROM, so the RFID tool can now be used in a stand-alone operation.

Replay and relay attacks are possible and are now displayed on the computer screen in an understandable manner. Through analyzing and replaying the data it is very easy to fool a reader if only the ISO protocol is in use. A challenge-and-response protocol can ensure security against such replay attacks.

There is no security against relay attacks, so it is inadvisable to use RFID cards in a security sensitive area. If the RFID tool is able to channel the data over distances, e.g. with WLAN (wireless local area network) chips, the genuine reader and the genuine tag could be several meters away from each other. The attacker can also analyze the relayed data afterwards.

References

- [1] Wikipedia - Radio Frequency Identification
http://en.wikipedia.org/wiki/Radio-frequency_identification
- [2] Timo Kasper - Embedded Security Analysis of RFID Devices
<http://www.crypto.ruhr-uni-bochum.de/theses.html>
- [3] RFID Journal - The History of RFID Technology
<http://www.rfidjournal.com/article/articleview/1338/1/129/>
- [4] Hitachi - The World's Smallest RFID IC
<http://www.hitachi.co.jp/Prod/mu-chip/index.html>
- [5] WinAVR - AVR-GCC for Windows
<http://winavr.sourceforge.net/>
- [6] ISO/IEC 14443. Identification Cards
Contactless Integrated Circuit(s) Cards
Proximity Cards - Part 1-4
<http://www.iso.ch>
- [7] AVRDUDE - AVR Downloader/UploaDEr
<http://www.nongnu.org/avrdude/>
- [8] Philips Mifare - Contactless Smart Cards
<http://www.mifare.net>
- [9] Microsoft Visual C++ 2005 Express Edition
<http://msdn.microsoft.com/vstudio/express/visualc/>
- [10] der-Hammer: Hterm - A Terminal Program for Windows
<http://www.der-hammer.info/terminal/>
- [11] Wikipedia - Cyclic Redundancy Check
http://en.wikipedia.org/wiki/Cyclic_redundancy_check
- [12] Wikipedia - HyperTerminal
<http://en.wikipedia.org/wiki/Hyperterminal>

- [13] FTDI - Future Technology Devices International Ltd.
FT245R - USB FIFO IC
<http://www.ftdichip.com/Products/FT245R.htm>
- [14] Atmel ATMega32
http://www.atmel.com/dyn/products/product_card.asp?family_id=607&family_name=AVR+8%2DBit+RISC+&part_id=2014
- [15] Microsoft Windows SDK
<http://www.microsoft.com/downloads/details.aspx?FamilyId=A55B6B43-E24F-4EA3-A93E-40C0EC4F68E5&displaylang=en>
- [16] EM Electronic - EM4094
13.56MHz, compliant ISO 14443A&B, Analog Front End IC for RFID Base Station
<http://www.emmicroelectronic.com/Products.asp?IdProduct=203>