Design comparison between HiperLAN/2 and IEEE802.11a services

Emil Edbom
Henrik Henriksson

2001-12-14
Design comparison between HiperLAN/2 and IEEE802.11a services

Examensarbete utfört i trådlös kommunikation vid Tekniska Högskolan i Linköping, Campus Norrköping

Emil Edbom
Henrik Henriksson

Handledare: Fredrik Olsson
Examinator: Qin-Zhong Ye

Norrköping den 14 december 2001
Titel
Design comparison between HiperLAN/2 and IEEE802.11a services

Sammanfattning
Abstract
This paper is a study and comparison between the two Wireless LAN (WLAN) standards HiperLAN/2 and IEEE 802.11a. WLANs are used instead or together with ordinary LANs to increase mobility in for example an office. HiperLAN/2 is an European standard developed by ETSI and the IEEEs standard is American.
A WLAN-card consists roughly of a Medium Access Control (MAC), Physical layer (PHY) and an antenna. The antenna is the same for the different standards.
Both standards operates at 5.4 GHz with a maximum transmission rate at 54 Mbit/s and they use OFDM to modulate the signal. This means that the physical layer in the two standards is similar.
The differences between the standards are in the Medium Access Control (MAC) layer. HiperLAN/2 has a much more complex MAC since it is developed with the starting point in cellular phones. Therefore this MAC is not very similar to ETHERNET that is the protocol used by regular network. On the other hand it is built to be compatible with cellular phones and other applications.
The 802.11a MAC is very much the same as in the 802.11b standard that is the most used standard at present. The difference is that 802.11a can send at much higher data rates. This MAC is build with starting point in ETHERNET so it has a similar interface to the computer. This makes it less complex.
The different MACs can provide different services. The greatest difference is that 802.11a can use a distributed send mode where any STA can send if the medium is idle. This reminds a lot of ETHERNET but they use different methods to sense if the medium is idle. In HiperLAN/2 are all transmissions scheduled by the AP. 802.11a can operate in a similar way but at the moment this mode is not as fully developed as in HiperLAN/2.
There are working groups in IEEE that works toward an improvement of 802.11a so it can use queues with different priorities, this is already implemented in HiperLAN/2.
Another important issue in wireless environment is security. Both standards use encryption to protect their messages. The difference is that HiperLAN/2 changes their encryption key for every connection where 802.11a uses the same key the whole time. This gives HiperLAN/2 a better security with todays standard but there are working groups dealing with implementing key-exchange functions and Kerberos use in 802.11a.
Chapter 8 is a description of a program that we developed in C++. The program is used to monitor the different registers and ports a WLAN-card use. It is written for a 802.11b card and should be used together with Windows 2000. The source code can be found in appendix C.

Nyckelord
IEEE802.11a
HiperLAN/2
Wireless LAN
WLAN
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Preface</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Introduction to Wireless Local Area Network</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Benefits and disadvantages</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Mobility</td>
<td>4</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Cheap and simple installations</td>
<td>4</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Interference</td>
<td>4</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Security</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Components of a WLAN</td>
<td>5</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Mobile Terminal/Station</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Access Point</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>System overview</td>
<td>6</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Antenna</td>
<td>7</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Radio</td>
<td>7</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Baseband</td>
<td>7</td>
</tr>
<tr>
<td>2.3.4</td>
<td>MAC- Medium access control</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>Signal path from application to antenna</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>IEEE 802.11a</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Protocol stack</td>
<td>11</td>
</tr>
<tr>
<td>3.1.1</td>
<td>LLC – Logical Link Control</td>
<td>11</td>
</tr>
<tr>
<td>3.1.2</td>
<td>MAC</td>
<td>11</td>
</tr>
<tr>
<td>3.1.2.1</td>
<td>General frame format</td>
<td>12</td>
</tr>
<tr>
<td>3.1.2.2</td>
<td>Specific frames</td>
<td>15</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Physical Layer Convergence Procedure (PLCP)</td>
<td>20</td>
</tr>
<tr>
<td>3.1.3.1</td>
<td>Preamble</td>
<td>20</td>
</tr>
<tr>
<td>3.1.3.2</td>
<td>SIGNAL</td>
<td>21</td>
</tr>
<tr>
<td>3.1.3.3</td>
<td>DATA</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>Hiperlan/2</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>Protocol stack</td>
<td>23</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Physical layer</td>
<td>24</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Data Link Control layer</td>
<td>25</td>
</tr>
<tr>
<td>4.1.2.1</td>
<td>Error Control (EC)</td>
<td>25</td>
</tr>
<tr>
<td>4.1.2.2</td>
<td>Radio Link Control (RLC)</td>
<td>26</td>
</tr>
<tr>
<td>4.1.2.3</td>
<td>Medium Access Control layer (MAC)</td>
<td>27</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Convergence layer</td>
<td>32</td>
</tr>
<tr>
<td>4.1.3.1</td>
<td>Ethernet Convergence layer</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>Quality of Service (QoS)</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>802.11a</td>
<td>36</td>
</tr>
<tr>
<td>5.1.1</td>
<td>DCF</td>
<td>37</td>
</tr>
<tr>
<td>5.1.1.1</td>
<td>Interframe Space (IFS)</td>
<td>37</td>
</tr>
<tr>
<td>5.1.1.2</td>
<td>Random Backoff Time</td>
<td>38</td>
</tr>
<tr>
<td>5.1.1.3</td>
<td>Transfer procedure</td>
<td>39</td>
</tr>
<tr>
<td>5.1.2</td>
<td>PCF</td>
<td>40</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Enhanced Cooperation Functions</td>
<td>40</td>
</tr>
<tr>
<td>5.1.3.1</td>
<td>EDCF</td>
<td>40</td>
</tr>
<tr>
<td>5.1.3.2</td>
<td>EPCF</td>
<td>41</td>
</tr>
<tr>
<td>5.1.3.3</td>
<td>HCF</td>
<td>41</td>
</tr>
<tr>
<td>5.2</td>
<td>HiperLAN/2</td>
<td>41</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Resource reservation</td>
<td>42</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Priority in Ethernet Convergence Layer</td>
<td>42</td>
</tr>
<tr>
<td>6</td>
<td>Security</td>
<td>44</td>
</tr>
</tbody>
</table>
1 Preface

According to the market's demands of high mobility and high transmission speeds, there has been a growth of new standards for Wireless Local Area Networks (WLANs). The standards are:

IEEE 802.11a developed by the Institute of Electrical and Electronics Engineers (IEEE)
HiperLAN/2 developed by the European Telecommunications Standards Institute (ETSI)

Both standards are operating at 5.4 GHz and that is in a band that is free of licences.

802.11a is an improvement of 802.11b (Wi-Fi) that is the dominating standard for WLANs operating at 2.4 GHz. This standard is built with considerations of Ethernet that is the most common technique in LANs.

HiperLAN/2 has its roots in telecommunication techniques but it is also compliant with Ethernet.

Since it probably will be different standards in Europe and USA, we decided to examine the differences and similarities between them. We study all the digital parts in the transceiver, but we have been concentrating our work to the Medium Access Control (MAC) layer and surrounding layers since most of the differences are there.

We also tried to figure out if it is possible and if so, how to combine the two standards.

The second part of the thesis work is a program written in C++ that monitors the registers and ports a IEEE 802.11b card uses. This program was used as a foundation to develop a 802.11a compliant program but since no such card was available, the program only compliant with 802.11b.
2 Introduction to Wireless Local Area Network

Wireless Local Area Network (WLAN) is an alternative to an ordinary LAN where the data is transmitted through cables or fibres. In a WLAN the air is used as the transmitting medium. It works in a similar way as a LAN but the transmission rate is lower. One way to increase the transmission rate is to transmit at a higher frequency and by that gaining bandwidth. The new standards are IEEE’s 802.11a and ETSI’s Broadband Radio Access Networks (BRAN) project High PERformance Local Area Network type 2 (HiperLAN/2) both operate at 5.4 GHz, which is more than twice as high frequency as today’s WLANs at 2.4 GHz.

2.1 Benefits and disadvantages

There are some large benefits using a WLAN but there are also some disadvantages.

2.1.1 Mobility

With a WLAN you are able to physically move around while using an appliance connected to the network. This makes an office more variable. If two WLANs are compliant you can bring your laptop from one to another and immediately get connected to the new one. This is very convenient when you travel, you bring your laptop and when you arrive to for example an airport you are able to get your emails while you wait for your bags. It is also very practical for people using their computer outside the network all day. They only have to walk into the office and upload their work. [1]

2.1.2 Cheap and simple installations

Since a WLAN does not use cables it can be implemented in environments where an ordinary LAN causes trouble. The wireless link can connect two buildings even if there is a road or a river separating them. With a cable between the buildings it will be exposed to wear and one of the primary faults in a network are cable faults. This means that the installation in difficult-to-wire areas becomes both easy and in the long run cheaper. In more conventional environments such as offices a WLAN can be very convenient. Many companies reorganize, expand or make renovations that involve recabling the network. With a WLAN you can move around as much as you want as long as the network is within reach. If you is out of reach from your network the system tries to detect a network that fulfils your demands. This might happen if you move between different department at work. This is called roaming and it is also used to gain best transmission rate. If your computer is connected to one network but finds out that a better network is at hand it switches without distortion. It is also possible to use WLAN and LAN together in a seamless manner. [1]

2.1.3 Interference

Since the transmission is through the air there is a risk of interference from products using similar radio frequencies. There are two types of interference:

- Inward Interference is when the WLAN is disturbed by other products or transmitting systems.
- Outward Interference is when the WLAN disturbs other products or transmitting systems.
2.1.4 Security
The main security problem for WLANs is that they intentionally propagate data over areas the company does not physically control. Radio waves can for instance penetrate building walls and then be receivable for others. The problem is much smaller using Ethernet since the eavesdropper has to listen close to the cable. However in a WLAN it is possible to use a network access code to prevent intruders from eavesdropping. It is optional to use the code depending on which security-level the user demand.

Another problem is the risk of electronic sabotage, in which someone jams the network and keeps you from using the network. It is easy to jam the network if you have a wireless product of the same manufacture as the network you want to jam. You just keep resending packages all the time. Then all other would have to wait.

To avoid eavesdropping and jamming many vendors require you to establish a network access code and set the code in every workstation.

One additional option is to use encryption of the data. Both HiperLAN/2 and 802.11a uses a 64-bit encryption key. This is called “Shared Key Authentication” in 802.11a and “pre shared key” in HiperLAN/2.

2.2 Components of a WLAN
A WLAN is built of Access Points (AP) and Mobile Terminals/Stations (MT/STA). HiperLAN/2 has named them Mobile Terminals and 802.11a names them Stations. From now on in this text they are denoted MT/STA. Several MT/STAs can associate to an AP at the same time. This configuration is called a BSS (Basic Service Set) in 802.11a and Radio Cell in HiperLAN/2. In this text they are from now on denoted BSS. If an AP accesses another AP they form an ESS (Extended Service Set). The Distribution System (DS) is used to interconnect different BSSs and integrated LANs to form an ESS (called Radio Access Network in HiperLAN/2).

In Figure 1 BSS 1 and BSS 2 is connected to each other using APs and the DS. If you want to connect your WLAN to a wired LAN you can use a portal. The portal provides logical integration between the 802.11a and existing wired LANs. Most APs today includes the function of a portal. The portal is called Convergence Layer in HiperLAN/2.
2.2.1 Mobile Terminal/Station
A MT/STA is an end-user device such as:

- Desktop computer
- Laptop computer
- Palmtop computer
- Handheld printers

Several STAs can connect as an ad hoc network and is then called IBSS (Independent Basic Service Set). This can also be done in HiperLAN/2 but then must one controller be a Central Controller (CC) that works similar as an AP. This is since HiperLAN/2 need a unit that controls the transmissions in the network. [1, 7, 8]

2.2.2 Access Point
To be able to connect your IBSS to for example a wired LAN you will need an AP. Data move between a BSS and the DS via an AP. An AP is a STA that provides access to the DS at addition to acting as a STA. In HiperLAN/2 is the AP very important since HiperLAN/2 is centrally organized. That means that the AP controls all transmissions in the network and determines which MT that is allowed to send.

2.3 System overview
Both APs and MT/STAs have a similar architecture in both 802.11a and HiperLAN/2. Here follows a brief overview of the important parts in figure 2.
2.3.1 Antenna
An antenna optimised to transmit and receive at certain frequency interval. In USA it is 5.15 - 5.25, 5.25 – 5.35 and 5.725 – 5.825 GHz and in Europe 5.15 – 5.35 and 5.470 – 5.725 GHz.

2.3.2 Radio
The radio consists of amplifiers, filters, a switch and a radio chip. The radio chip contains D/A- and A/D-converters, filters and a mixer, see figure 3.

2.3.3 Baseband
Both 802.11a and HiperLAN/2 uses OFDM (Orthogonal Frequency Division Multiplexing) to transceive signals. They modulate the 48 sub carriers using either Binary or Quadrature Phase Shift Keying (BPSK or QPSK), 16 – Quadrature Amplitude Modulation (16-QAM) or 64-QAM depending on which data rate they want to transceive at. [4]

In table 1 is the different data-rates and modulation techniques listed. [5]
### Table 1: Data rates and modulation techniques

<table>
<thead>
<tr>
<th>Modulation</th>
<th>Code Rate</th>
<th>Net rate</th>
<th>Byte per Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPSK</td>
<td>1/2</td>
<td>6 Mbit/s</td>
<td>3</td>
</tr>
<tr>
<td>BPSK</td>
<td>3/4</td>
<td>9 Mbit/s</td>
<td>4.5</td>
</tr>
<tr>
<td>QPSK</td>
<td>1/2</td>
<td>12 Mbit/s</td>
<td>6</td>
</tr>
<tr>
<td>QPSK</td>
<td>3/4</td>
<td>18 Mbit/s</td>
<td>9</td>
</tr>
<tr>
<td>16QAM</td>
<td>3/4</td>
<td>36 Mbit/s</td>
<td>18</td>
</tr>
<tr>
<td>64QAM</td>
<td>3/4</td>
<td>54 Mbit/s</td>
<td>27</td>
</tr>
<tr>
<td><strong>HiperLAN/2 only</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-QAM</td>
<td>9/16</td>
<td>27 Mbit/s</td>
<td>13.5</td>
</tr>
<tr>
<td><strong>IEEE 802.11a only</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-QAM</td>
<td>1/2</td>
<td>24 Mbit/s</td>
<td>12</td>
</tr>
<tr>
<td>64QAM</td>
<td>2/3</td>
<td>48 Mbit/s</td>
<td>24</td>
</tr>
</tbody>
</table>

The baseband also scrambles and encodes the signals to enable OFDM.

### 2.3.4 MAC- Medium access control

The MAC-layer provides functions such as accessing the wireless medium, joining a network and provides authentication and security.

Most of the differences between 802.11a and HiperLAN/2 are in the MAC-layer. They are described in detail later on in this paper.

### 2.4 Signal path from application to antenna

This is a simple description of how a signal travels from the application to the antenna. It is a very brief description so it does not involve much theory of for example the modulation process.

The signal from the application to the MAC is a bit stream containing data and addresses. The MAC put the stream in one or several frames and sends the frames to the Physical layer.

![101110100101001]

*Figure 4: One octet is chosen to the example*

The Physical layer uses OFDM to code and modulate the bits. In this example BPSK is used with a code rate of 0.5 since it is the simplest form. In the real system OFDM uses 48 sub carriers to gain better throughput despite the long transmission-time (four microseconds). The long transmission-time is used to assure that one OFDM-symbol has arrived before the next one is sent. This example only uses 4 sub carriers to show the principle of OFDM.

The coding is done according to figure 5 and it is used to spread the information over a wider range of bits.
Figure 5: Creating of the output streams X and Y

If the last four bits in the frame in figure 4 is coded the result is (the rightmost bit transmitted first):

Input: 0101
X: 1001
Y: 0011
They are put together to 01001011 according to Y3, X3…Y0, X0.

If another bit rate is desirable you remove bits according to table 2.

<table>
<thead>
<tr>
<th>Code Rates r</th>
<th>Puncturing pattern</th>
<th>Remaining bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2</td>
<td>Xdo: 1 Ydo: 1</td>
<td>X0 Y0</td>
</tr>
<tr>
<td>9/16</td>
<td>Xdo: 1 1 1 1 1 1 1 0 Ydo: 1 1 1 1 0 1 1 1</td>
<td>X0 Y0 X1 Y1 X2 Y2 X3 Y3 X4 Y4 X5 X6 X7 Y7 Y8</td>
</tr>
<tr>
<td>3/4</td>
<td>Xdo: 1 1 0 Ydo: 1 0 1</td>
<td>X0 Y0 X1 Y2</td>
</tr>
<tr>
<td>2/3</td>
<td>Xdo: 1 1 1 1 1 Ydo: 1 0 1 0 1 0</td>
<td>X0 Y0 X1 X2 Y2 X3 X4 Y4 X5</td>
</tr>
</tbody>
</table>

Table 2: Different code rates and puncturing pattern

BPSK means that there are two different phases in the transmission. When the bit changes from zero to one the phase changes 180 degrees. If the last four bits in figure 4 is coded and transmitted on four sub carriers the signals in figure 6 is sent.
Every sub carrier is assigned a certain bandwidth of frequency. Two sub carriers are divided with 0.3125 MHz and with 52 carriers the total bandwidth is 16.25 MHz. 48 of the carriers contains data and the last four is for reference information. The bandwidth is used more efficient with OFDM since it allows overlapping between different carriers to save bandwidth.

The sub carriers are sent at a carrier frequency of 5.4 GHz.
3 IEEE 802.11a

In November 1997 the IEEE 802.11-standard was finished. 802.11a is an extension of this standard and it was finished in 1999. The extension offers higher transmission rates and improved Quality of Service (QoS). The standard builds on other IEEE-standards, for example IEEE 802.2 LOGICAL LINK CONTROL and 802.3 CSMA/CD (ETHERNET).

3.1 Protocol stack

The protocol stack of an application is the hierarchy of the different theoretical layers in the application. The layers do not match the physical structure in figure 8 exactly but you can see some similarities.

![Diagram of the protocol stack](image)

Figure 8: The different layers in 802.11a and the LLC layer

3.1.1 LLC – Logical Link Control

The LLC is not really a part of 802.11a but since it is the highest layer of the IEEE 802 Reference Model it is included briefly. It provides addressing and data link control and it is independent of the topology, transmission medium and medium access control chosen. [1]

LLC and MAC is communicating through sending indication-, request- and status-primitives. Since 802.11 uses a similar structure as 802.3 (ETHERNET) is the communication between the different layers relative easy.

3.1.2 MAC

Since the MAC sub layer is responsible for channel allocation procedures, Protocol Data Unit (PDU) addressing, frame formatting, error checking and fragmentation it contains a lot of information. Each MAC-frame consists of following components:

- Header, containing frame control, duration, address and sequence control information
- Body of variable length containing information specific to the frame type
- Frame Check Sequence (FCS), a 32-bit Cyclic Redundancy Code (CRC) [8]

There are three different types of frames in 802.11a: management, control and data.
The management frame is used for association, disassociation, authentication deauthentication, timing and synchronisation.

The control frame is used for handshaking in the Contention Period (CP), for Acknowledgement in the CP and at the end of the Contention Free Period (CFP).

The Data frames are used for transmission of data.

3.1.2.1 General frame format
The MAC Protocol Data Units (MPDU) or frames are described as a sequence of fields in a specific order.

Figure 9: The general frame format of a MPDU

3.1.2.1.1 Frame Control
The Frame control field is a part of the MAC-header and it consists of following fields.

Figure 10: Frame control field

3.1.2.1.1.1 Protocol version field
The protocol version is set to zero for this standard. It is implemented for future usage. The level will only be incremented when a fundamental incompatibility exists between a new revision and the prior edition of the standard.

3.1.2.1.1.2 Type and Subtype fields
The Type and The Subtype fields define together the function of the frame. Each of the three different types has several subtypes as can be seen in table 3.
<table>
<thead>
<tr>
<th>Type value</th>
<th>Type description</th>
<th>Subtype value</th>
<th>Subtype description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b3 b2</td>
<td></td>
<td>b7 b6 b5 b4</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>Management</td>
<td>0000</td>
<td>Association request</td>
</tr>
<tr>
<td>00</td>
<td>Management</td>
<td>0001</td>
<td>Association response</td>
</tr>
<tr>
<td>00</td>
<td>Management</td>
<td>0010</td>
<td>Reassociation request</td>
</tr>
<tr>
<td>00</td>
<td>Management</td>
<td>0011</td>
<td>Reassociation response</td>
</tr>
<tr>
<td>00</td>
<td>Management</td>
<td>0100</td>
<td>Probe request</td>
</tr>
<tr>
<td>00</td>
<td>Management</td>
<td>0101</td>
<td>Probe response</td>
</tr>
<tr>
<td>00</td>
<td>Management</td>
<td>0110-0111</td>
<td>Reserved</td>
</tr>
<tr>
<td>00</td>
<td>Management</td>
<td>1000</td>
<td>Beacon</td>
</tr>
<tr>
<td>00</td>
<td>Management</td>
<td>1001</td>
<td>Announcement traffic indication message (ATIM)</td>
</tr>
<tr>
<td>00</td>
<td>Management</td>
<td>1010</td>
<td>Disassociation</td>
</tr>
<tr>
<td>00</td>
<td>Management</td>
<td>1011</td>
<td>Authentication</td>
</tr>
<tr>
<td>00</td>
<td>Management</td>
<td>1100</td>
<td>Deauthentication</td>
</tr>
<tr>
<td>00</td>
<td>Management</td>
<td>1101-1111</td>
<td>Reserved</td>
</tr>
<tr>
<td>01</td>
<td>Control</td>
<td>0000-1001</td>
<td>Reserved</td>
</tr>
<tr>
<td>01</td>
<td>Control</td>
<td>1010</td>
<td>Power Save (PS)-Poll</td>
</tr>
<tr>
<td>01</td>
<td>Control</td>
<td>1011</td>
<td>Request To Send (RTS)</td>
</tr>
<tr>
<td>01</td>
<td>Control</td>
<td>1100</td>
<td>Clear To Send (CTS)</td>
</tr>
<tr>
<td>01</td>
<td>Control</td>
<td>1101</td>
<td>Acknowledgment (ACK)</td>
</tr>
<tr>
<td>01</td>
<td>Control</td>
<td>1110</td>
<td>Contention Free(CF)-End</td>
</tr>
<tr>
<td>01</td>
<td>Control</td>
<td>1111</td>
<td>CF-End + CF-Ack</td>
</tr>
<tr>
<td>10</td>
<td>Data</td>
<td>0000</td>
<td>Data</td>
</tr>
<tr>
<td>10</td>
<td>Data</td>
<td>0001</td>
<td>Data + CF-Ack</td>
</tr>
<tr>
<td>10</td>
<td>Data</td>
<td>0010</td>
<td>Data + CF-Poll</td>
</tr>
<tr>
<td>10</td>
<td>Data</td>
<td>0011</td>
<td>Data + CF-Ack + CF-Poll</td>
</tr>
<tr>
<td>10</td>
<td>Data</td>
<td>0100</td>
<td>Null function (no data)</td>
</tr>
<tr>
<td>10</td>
<td>Data</td>
<td>0101</td>
<td>CF-Ack (no data)</td>
</tr>
<tr>
<td>10</td>
<td>Data</td>
<td>0110</td>
<td>CF-Poll (no data)</td>
</tr>
<tr>
<td>10</td>
<td>Data</td>
<td>0111</td>
<td>CF-Ack + CF-Poll (no data)</td>
</tr>
<tr>
<td>10</td>
<td>Data</td>
<td>1000–1111</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>Reserved</td>
<td>0000–1111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 3: Frame types and subtypes

3.1.2.1.1.3 To/From DS field

The field To DS is set to 1 in data type frames destined for the DS. The From DS field is set to 1 in data type frames exiting the DS. See table 4 for combinations of To/From DS.

<table>
<thead>
<tr>
<th>To/From DS values</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>To DS = 0</td>
<td>A data frame direct from one STA to another STA within the same IBSS, as well as all management and control type frames.</td>
</tr>
<tr>
<td>From DS = 0</td>
<td>Data frame destined for the DS.</td>
</tr>
<tr>
<td>To DS = 1</td>
<td>Data frame exiting the DS.</td>
</tr>
<tr>
<td>From DS = 1</td>
<td>Wireless distribution system (WDS) frame being distributed from one AP to another AP.</td>
</tr>
</tbody>
</table>

Table 4: To/From DS combinations in data type frames

3.1.2.1.1.4 More Fragments field

Indicates if there are more fragments of the current MSDU or MMPDU to be sent. A fragment is a part of a MSDU or MMPDU. It is efficient to have messages divided into fragments if an error occurs in a transmission. It is then only necessary to retransmit the fragment and not the whole message. The size of the fragments can be adjusted by the user to fit the environment where the WLAN is oriented. It is better to send small fragments in an area with high distortion and vice versa.
3.1.2.1.1.5 **Retry field**
This field is set to 1 in any data or management type frame that is a retransmission of an earlier frame. This is used to avoid duplicate frames when resending erroneous frames.

3.1.2.1.1.6 **Power Management field**
This is an indication of what mode the transmitting station will be after the successful completion of the frame exchange sequence. If it is set to 1 then the station will be in Power-Save mode (PS). When a station is in PS it is not able to transmit or receive and it consumes very low power. The AP buffers frames that are directed to an STA in PS. Otherwise it is in active mode which means that it can transmit and receive continuously. It is always set to 1 if an AP is transmitting the frame.

3.1.2.1.1.7 **More Data field**
The More Data field is used to indicate to a STA in power-save mode that more MSDUs, or MMPDUs are buffered for the STA at the AP.

3.1.2.1.1.8 **WEP field**
This bit is set to 1 if the Frame Body field has been processed by the WEP-algorithm. It is only the Frame Body that is encrypted and the field can only be set to 1 in frames of type Data or type Management and subtype Authentication.

3.1.2.1.1.9 **Order**
The Order field bit is set to 1 in any Data type frame being sent using the Strictly-Ordered service class. This class is used when a higher-layer protocol do not accept that the MAC layer reorders the frames transmitted to achieve higher likelihood of success.

3.1.2.1.2 **Duration/ID field**
The Duration/ID field is 16 bits in length. The contents of this field are as follows:

- In control frames of subtype Power Save (PS)-Poll, the Duration/ID field contains the Association Identity (AID) of the station that transmitted the frame in the 14 least significant bits (lsb), with the 2 most significant bits (msb) both set to 1. The AID is in the range 1–2007 and is given to the STA during the association process (see 3.1.2.2.3.5).

- Otherwise the Duration/ID field contains a duration value that is defined for each frame type. In the contention-free period (CFP), the duration field is set to 32 768.

Whenever the contents of the Duration/ID field are less than 32 768, the duration value is used to update the network allocation vector (NAV) in each STA. The NAV is used to prevent collisions in the CP. It is decreased every microsecond and no STA can access the medium when NAV > 0.

3.1.2.1.3 **Address fields**
There are four address-fields in the MAC frame format but some frames do not contain all of them. They contain different addresses depending on the To-DS and From-Ds bits. Possible addresses are Destination Address (DA), Source Address (SA), Receiver Address (RA), Transmitter Address (TA) and Basic Service Set IDentifier (BSSID). Each address field contains a 48-bit MAC address.
3.1.2.1.4 Sequence Control field
This 16-bit field is divided into two sub fields, the Sequence Number and the Fragment Number. Each Sequence can be divided into 16 fragments.

The Sequence Number field is a 12-bit field indicating the sequence number of an MSDU or MMPDU. Each MSDU or MMPDU transmitted by a STA is assigned a sequence number from 0 to 4096 assigned from a single modulo 4096-counter. Every fragment of an MSDU or MMPDU contains the assigned sequence number. The sequence number remains constant during all retransmissions of an MSDU, MMPDU, or fragment thereof.

The Fragment Number field is 4-bit in length and it indicates the number of each fragment of an MSDU or MMPDU. It is set to zero in the first or only fragment of an MSDU or MMPDU and is incremented by one for each successive fragment of that MSDU or MMPDU. It remains constant in all retransmissions of the fragment.

3.1.2.1.5 Frame Body
The Frame Body is of variable length (0 – 2312 bytes) and it contains data and information specific to the individual type and subtype of a frame.

3.1.2.1.6 Frame Check Sequence (FCS)
This field is 32 bits in length and it contains the result after applying a 32-bit Cyclic Redundancy Code -polynomial (CRC) on the header and body of the MAC. This polynomial is used in other 802-standards as well i.e. Ethernet. It is used to check for errors in the transmission. [2]

3.1.2.2 Specific frames
There are three different sort of frame formats used for different purposes. They are Control, Data and Management frames.

3.1.2.2.1 Control frames
These frames are used for handshaking in the Contention Period (CP), for Acknowledgement in the CP and at the end of the Contention Free Period (CFP).

3.1.2.2.1.1 Request To Send (RTS)
This is the first frame in the frame exchange handshake between transmitter and receiver. RTS is sent from the STA to the AP when the STA wants to send a data or management frame that is bigger than the optional value RTS_threshold. The RA is the address to the STA that should receive the frame and the TA is the address of the frame transmitting the RTS frame.

The duration value is the time required for the whole transmission. [8]
3.1.2.2.1.2 Clear To Send (CTS)
When receiving a RTS the AP should respond with CTS in a predefined time called Short InterFrame Space (SIFS) if the transmission is allowed. You can read more about SIFS in 5.1.1.1.1. The RA is copied from TA in the RTS and the duration value is the value in RTS minus the time required to transmit the CTS and one SIFS. [8]

![Figure 12: Clear to send frame](image)

3.1.2.2.1.3 Acknowledgement (ACK)
After sending the data or management frame the acknowledgement frame is sent. This is the fourth and last frame in the frame exchange handshake. It is sent to inform the sender if the transmission was successful. [2]

3.1.2.2.1.4 Power-Save Poll (P-S Poll)
The P-S Poll frame is sent from a STA that has been in power-save mode. It is sent to check if the AP has any frames buffered that should be transmitted to the STA. The duration/ID is in this frame the Association Identifier (AID) that the STA got during the association process. BSSID identifies the AP and TA the STA. The AP answers with either the data that was buffered or the Null function. (see 3.1.2.2.2 Data Frames) [2]

![Figure 13: Power save poll frame](image)

3.1.2.2.1.5 CF-End and CF-End + CF-Ack
The CF-End frame and CF-End + CF-Ack frame are used to indicate that the Contention Free Period (CFP) is over and that it is free to compete for access to the medium. The latter of the two also acknowledge the last transmission. RA is the group address since this frame should go out to all STAs in the BSS. BSSID is the AP MAC-address. The duration value is zero to set the NAV in the STAs to zero and thereby allowing them to send.

![Figure 14: CF-End and CF-End + CF-Ack frame](image)
3.1.2.2 Data frames

The data frame looks like the general frame format in fig 9. The format is independent of the subtype but the content of the address field changes with the changes in To DS and From DS. The data frame has variable length with a minimum of 29 bytes and a maximum of 2346 bytes. There are eight different frame types divided into two groups, containing data and no data.

3.1.2.2.1 Data

The four data frames containing data are Data, Data + CF-Ack, Data + CF-Poll and Data + CF-Ack + CF-Poll.

The Data + CF-Ack is identical to Data except that it also acknowledge the previous transfer. It may only be sent during CFP and it is never used in an IBSS.

The Data + CF-Poll is identical to Data except that the AP also tells the STA to send any buffered frames directly after receiving the frame. It may only be sent during CFP and it is never used in an IBSS.

Data + CF-Ack + CF-Poll is a combination of the both previous frames.

3.1.2.2.2 No Data

The frames containing no data build on the Null Function frame. It is used to carry the power management bit to the AP when a STA changes to a low power operating state. The other three frames are equal as the three extensions of data above: CF-Ack, CF-Poll and CF-Ack + CF-Poll.

3.1.2.2.3 Management frames

The management frames frame format is independent of subtype. The difference between different frames is the frame body and the BSSID. The frame body consists of fixed fields and information elements. These are defined for each subclass and are mandatory unless others are stated.

![Figure 15: Fields in the management frames](image)

3.1.2.2.3.1 Fixed Fields

The fixed fields are of fixed length and they are all mandatory.

3.1.2.2.3.1.1 Authentication Algorithm Number Field

This field is 2 octets long and it can be either 1 or 0. 0 if the system uses Open System authentication and 1 if Shared Key is used.

3.1.2.2.3.1.2 Authentication Transaction Sequence Number Field

This field is 2 octets and it indicates the current state of progress through a multistep transaction.

3.1.2.2.3.1.3 Beacon Interval Field

Beacon interval is the period of beacon transmissions measured in Time Units (TU) of 1024 microseconds. This field is also 2 octets long.
3.1.2.3.1.4 Capability information field

This field consists of several sub fields that contains information about requested or advertised capabilities. The sub fields are ESS, IBSS, CF Pollable, CF Poll Request and Privacy. The remaining of the 2 octets is reserved, see figure 16.

<table>
<thead>
<tr>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESS</td>
<td>IBSS</td>
<td>CF Pollable</td>
<td>CF Poll Request</td>
<td>Privacy</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Figure 16: Sub fields in the Capability information field

If an AP is present it has the ESS set to 1 and IBSS set to 0 in Beacon or Probe Response frames.

In an IBSS is the ESS set to 0 and IBSS to 1 when the STA sends Beacon or Probe Response frames.

A STA sets bits B2 and B3 in Association and Reassociation Request frames according to table 5.

<table>
<thead>
<tr>
<th>CF-Pollable B2</th>
<th>CF-Poll request B3</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>STA is not CF-Pollable</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>STA is CF-Pollable, not requesting to be placed on the CF-Polling list</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>STA is CF-Pollable, requesting to be placed on the CF-Polling list</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>STA is CF-Pollable, requesting never to be polled</td>
</tr>
</tbody>
</table>

Table 5: The meaning of different combinations of CF-Pollable and CF-Poll request in a STA

A AP sets bits B2 and B3 in Beacon, Probe Response, Association Response and Reassociation Response frames according to table 6.

The Association and Reassociation Response is set equal to the last Beacon or Probe Response frame transmitted.

<table>
<thead>
<tr>
<th>CF-Pollable B2</th>
<th>CF-Poll request B3</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No point coordinator at AP</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Point coordinator at AP for delivery only (no polling)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Point coordinator at AP for delivery and polling</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 6: The meaning of different combinations of CF-Pollable and CF-Poll request in an AP

3.1.2.3.1.5 Current AP Address Field
This field holds the MAC-address of the AP the STA is associated with for the moment. The length of the field is 6 octets.

3.1.2.3.1.6 Listen Interval Field
This field indicates to the AP how often a STA wakes up and listen to Beacon frames. It is 2 octets long and it is expressed in units of Beacon Interval.

3.1.2.3.1.7 Reason Code Field
This field is used to indicate why an unsolicited frame of Disassociation or Deauthentication was generated. The field is 2 octets long and the different codes are displayed in table 7.
<table>
<thead>
<tr>
<th>Reason code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>Unspecified reason</td>
</tr>
<tr>
<td>2</td>
<td>Previous authentication no longer valid</td>
</tr>
<tr>
<td>3</td>
<td>Deauthenticated because sending station is leaving (or has left) IBSS or ESS</td>
</tr>
<tr>
<td>4</td>
<td>Disassociated due to inactivity</td>
</tr>
<tr>
<td>5</td>
<td>Disassociated because AP is unable to handle all currently associated stations</td>
</tr>
<tr>
<td>6</td>
<td>Class 2 frame received from nonauthenticated station</td>
</tr>
<tr>
<td>7</td>
<td>Class 3 frame received from nonassociated station</td>
</tr>
<tr>
<td>8</td>
<td>Disassociated because sending station is leaving (or has left) BSS</td>
</tr>
<tr>
<td>9</td>
<td>Station requesting (re)association is not authenticated with responding station</td>
</tr>
<tr>
<td>10–65 535</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 7: The reason why a Disassociation or Deauthentication is generated

3.1.2.2.3.1.8 Association ID (AID) Field
This field is 2 octets long and it is the ID the AP assigns the STA during association.

3.1.2.2.3.1.9 Status Code Field
The length of this field is 2 octets and it is used to indicate the success or failure of an operation. The codes are listed in table 8.

<table>
<thead>
<tr>
<th>Status code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful</td>
</tr>
<tr>
<td>1</td>
<td>Unspecified failure</td>
</tr>
<tr>
<td>2-9</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>Cannot support all requested capabilities in the Capability Information field</td>
</tr>
<tr>
<td>11</td>
<td>Reassociation denied due to inability to confirm that association exists</td>
</tr>
<tr>
<td>12</td>
<td>Association denied due to reason outside the scope of this standard</td>
</tr>
<tr>
<td>13</td>
<td>Responding station does not support the specified authentication algorithm</td>
</tr>
<tr>
<td>14</td>
<td>Received an Authentication frame with authentication transaction sequence number out of expected sequence</td>
</tr>
<tr>
<td>15</td>
<td>Authentication rejected because of challenge failure</td>
</tr>
<tr>
<td>16</td>
<td>Authentication rejected due to timeout waiting for next frame in sequence</td>
</tr>
<tr>
<td>17</td>
<td>Association denied because AP is unable to handle additional associated stations</td>
</tr>
<tr>
<td>18</td>
<td>Association denied due to requesting station not supporting all of the data rates in the BSSBasicRateSet parameter</td>
</tr>
<tr>
<td>19–65 535</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 8: Meaning of the status code

3.1.2.2.3.1.10 Timestamp Field
The Timestamp field is used by the Timing Synchronization Function (TSF) to synchronize all STAs in a BSS. It is 6 octets long.

3.1.2.2.3.2 Information Elements

3.1.2.2.3.2.1 Probe request and response
The probe request is sent from a STA to find a BSS. It contains the Service Set Identification (SSID) and the supported rates. SSID indicate the identity of an ESS or IBSS. When a station receives a request it should answer with a response that contains the same fields as the beacon frame.

3.1.2.2.3.2.2 Authentication
This frame is used to conduct a multiframe exchange between stations that ultimately results in the verification of the identity of each station to the other. It contains three fixed fields: the authentication algorithm number, the authentication transaction sequence number and the status code. The only information element is the challenge text.
3.1.2.2.3.2.3 Deauthentication
This frame is sent when a station wants to terminate an authentication relationship. It only contains the reason code that is a fixed field.

3.1.2.2.3.2.4 Association request and response
This is the first thing a STA must do to enter a BSS. The STA scans the area looking for Beacon frames with the right SSID. When it senses a BSS it sends an association request to ask for an association with a BSS and then the response is returned. The association is necessary to let the Distribution System (DS) know which AP a STA belongs to. A STA can therefore only be associated with one AP at one time.

The request includes two fixed fields: capability information and listen interval. The information fields are the SSID and the supported rates.

In the response there are three fixed fields: capability information, status code and AID. There are only one information field and it is the supported rates.

3.1.2.2.3.2.5 Reassociation request and response
This frames is sent when a STA associated with one BSS wants to associate with another BSS with the same SSID. They look the same as the association request and respond accept that the request contains the present AP address as well.

3.1.2.2.3.2.6 Disassociation
This frame is sent when a station wants to terminate an association relationship. It only contains the reason code that is a fixed field.

3.1.2.2.3.3 Announcement Traffic Indication Message (ATIF)
This frame can only be sent from STAs in an IBSS to another STA that is in power-save mode. It indicates that the sending STA has traffic buffered.

3.1.3 Physical Layer Convergence Procedure (PLCP)
The PLCP maps a MAC Protocol Data Unit (MPDU) into a PLCP Protocol Data Unit (PPDU) and vice versa. This is done since the different layers use frames of different length and content. The PPDU is a MPDU with certain bit sequences added to it. The PSDU in figure 17 is the same as the MSDU. [1]

![Figure 17: The parts of a PSDU](image)

3.1.3.1 Preamble
The preamble contains 9 repetitions of a “short training sequence” used for Automatic Gain Control (AGC), antenna selection, timing and a rough frequency acquisition in the receiver. It also contains two repetitions of a “long training sequence” which are used for channel estimation and fine frequency acquisition in the receiver. [5]
3.1.3.2 SIGNAL

In the SIGNAL are the fields RATE, Reserved, LENGTH, Parity and Tail forming a single OFDM-symbol. They are always transmitted with the most robust combination of modulating and coding rate in this case BPSK and 6Mbit/s.

<table>
<thead>
<tr>
<th>RATE (4 bits)</th>
<th>LENGTH (12 bits)</th>
<th>SIGNAL TAIL (6 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
<td>R3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 18: The Signal field

SIGNAL forms with the SERVICE-field the PLCP-header.

3.1.3.2.1 RATE

This field denotes what rate the field SERVICE, PSDU, Tail and Pad, denoted DATA, should be transmitted with. The rates are: 6, 9, 12, 18, 24, 36, 48 and 54 Mbit/s.

3.1.3.2.2 Reserved

This bit it reserved for future use.

3.1.3.2.3 LENGTH

This is an unsigned 12-bit integer that indicates the number of octets that the MAC wants to transmit in the PSDU. The value is determined from the TXVECTOR issued with the PHY-TXSTART.request primitive sent from MAC to PLCP.

3.1.3.2.4 Parity

This bit shall be a positive parity (even parity) bit for the bits in RATE, Reserved and LENGTH.

3.1.3.2.5 Tail

All 6 bits in the tail should be set to zero.

3.1.3.3 DATA

The DATA field consists of the SERVICE, PSDU, Tail and Pad fields. All bits in the DATA field are scrambled.

3.1.3.3.1 SERVICE

This field consists of 16 bits where the last 9 bits is reserved for future use and set to zero. The first 7 bits are also set to zero and they are used to synchronise the descrambler in the receiver.

3.1.3.3.2 PSDU

This is the data that the MAC wants to send, called MPDU in the MAC-layer.
3.1.3.3.3 Tail
This is a 6-bit field that contains zeros. The field return the convolutional encoder to the “zero state” which improves the error probability of the convolutional decoder.

3.1.3.3.4 Pad
The DATA field shall be a multiple of \( N_{CBPS}\) (Coded Bits Per Symbol) bits in length, the number of coded bits in an OFDM symbol (48, 96, 192, or 288 bits).

To achieve that, you extend the message with pad bits so that it becomes a multiple of \( N_{DBPS}\). The number of pad bits, \( N_{PAD}\), are computed from the length of the PSDU (LENGTH), the number of OFDM symbols, \( N_{SYM}\); and the number of bits in the DATA field, \( N_{DATA}\); as follows:

\[
N_{SYM} = \text{Ceiling}\left(\frac{16 + 8 \cdot \text{LENGTH} + 6}{N_{DBPS}}\right)
\]

\[
N_{DATA} = N_{SYM} \cdot N_{DBPS}
\]

\[
N_{PAD} = N_{DATA} - (16 + 8 \cdot \text{LENGTH} + 6)
\]

The function ceiling(.) returns the smallest integer value greater than or equal to its argument value. The pad bits are set to zero and are scrambled with the rest of the bits in the DATA field. [8]
4 HiperLAN/2

The HiperLAN/2 specifications are developed by ETSI BRAN (European Telecommunications Standards Institute, Broadband Radio Access Networks).

HIPERLAN/2 is a flexible Radio LAN standard designed to provide high speed access (up to 54 Mbit/s at PHY layer) to a variety of networks including 3G mobile core networks, ATM networks and IP based networks, and also for private use as a wireless LAN system. Basic applications include data, voice and video, with specific QoS parameters taken into account.

4.1 Protocol stack

The HiperLAN/2 protocol stack is depicted in figure 1. It consists of the Physical layer (PHY) at the bottom, the Data Link Control layer (DLC) in the middle and the Convergence layer (CL) at the top [7].

![Figure 19. HiperLAN/2 protocol stack](image)

The Physical layer delivers a basic data transport function by providing means of a baseband modem and an RF part.

The DLC layer consists of the Error Control (EC) function, the Medium Access Control (MAC) function and the Radio Link Control (RLC) function. It is divided in the user data transport functions, located on the right hand side, and the control functions located on the left hand side of figure 19.

The Convergence layer adepts service requests from higher layer to the service offered by the DLC and convert higher layer packets (SDUs) with variable size to a fixed size used by the DLC.
4.1.1 Physical layer

The reference configuration of the transmission chain is shown in figure 20.

Figure 20: Reference configuration of transmitter

The physical layer uses burst to transmit DLC PDU trains between AP/CC and MT, which consists of a preamble part and a data part. This includes the following functional entities at transmitter. For more information, see [7]:

- configuring the transmission bit rate by choosing appropriate PHY mode
- scrambling the PDU train content
- encoding the scrambled bits according to the forward error correction set during PHY layer configuration
- interleaving the encoded bits at the transmitter by using the appropriate interleaving scheme for the selected PHY layer mode
- sub-carrier modulation by mapping the interleaved bits into modulation constellation points
- producing the complex base-band signal by OFDM modulation
- inserting pilot sub-carriers, appending appropriate preamble to the corresponding PDU train at the transmitter and building the PHY layer burst
- performing radio transmission by modulating the radio frequency carrier with the complex base-band signal at transmitter

The physical (PHY) layer of HIPERLAN/2 is based on the modulation scheme Orthogonal Frequency Division Multiplexing (OFDM). OFDM has been chosen due its highly dispersive channels. The channel spacing is 20 MHz and each channel uses 52 sub carriers, 48 for data and 4 for tracking the phase for coherent demodulation [7].

The physical layer provides several modulation and coding alternatives due to different interference situations and distance between MTs and APs. Using various signal alphabets for modulating the OFDM sub-carriers can vary the data rate ranging from 6 Mbit/s to 54 Mbit/s. BPSK, QPSK, 16QAM are used as mandatory modulation schemes, whereas 64QAM is optional for both AP/CC and MT. The mode dependent parameters are listed in table 9 [7].
<table>
<thead>
<tr>
<th>Mode</th>
<th>Modulation</th>
<th>Code rate</th>
<th>PHY bit rate</th>
<th>bytes/OFDM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BPSK</td>
<td>1/2</td>
<td>6 Mbps</td>
<td>3.0</td>
</tr>
<tr>
<td>2</td>
<td>BPSK</td>
<td>3/4</td>
<td>9 Mbps</td>
<td>4.5</td>
</tr>
<tr>
<td>3</td>
<td>QPSK</td>
<td>1/2</td>
<td>12 Mbps</td>
<td>6.0</td>
</tr>
<tr>
<td>4</td>
<td>QPSK</td>
<td>3/4</td>
<td>18 Mbps</td>
<td>9.0</td>
</tr>
<tr>
<td>5</td>
<td>16QAM</td>
<td>9/16</td>
<td>27 Mbps</td>
<td>13.5</td>
</tr>
<tr>
<td>6</td>
<td>16QAM</td>
<td>3/4</td>
<td>36 Mbps</td>
<td>18.0</td>
</tr>
<tr>
<td>7</td>
<td>64QAM</td>
<td>3/4</td>
<td>54 Mbps</td>
<td>27.0</td>
</tr>
</tbody>
</table>

Table 9 PHY mode for HIPERLAN/2

4.1.2 Data Link Control layer

The DLC is the logical link between the AP and MT. It includes functions for both medium access and transmission (user plan) as well as terminal/user and connection handling (control plane) [6]. It consists of the following sub layers:

- Error Control (EC)
- Radio Link Control (RLC)
- Medium Access Control (MAC)

4.1.2.1 Error Control (EC)

The EC checks if the user messages was received properly and is responsible for delivering them in correct order to the convergence layer (CL). To ensure the correct delivering order the EC adds a sequence number to the user message. The MT and AP shall support three error control modes.

- Acknowledged mode
- Repetition mode
- Unacknowledged mode

4.1.2.1.1 Acknowledged mode

In acknowledged mode the error control is based on an Automatic Repeat reQuest (ARQ) scheme. This means that the receiver sends a feedback message to the sender, providing information about which messages that have been received correctly. The sender can choose to retransmit the messages or tell the receiver to discard the message.

4.1.2.1.2 Repetition mode

Repetition mode is only used for broadcast messages. In repetition mode the sender transmits the same message an arbitrary number times, which is decided by the vendor. If an error has occurred during transmission the receiver will wait until it receives the message again. If it takes to long time until this is done the message will be discarded. No feedback information is sent from the receiver to the sender.

4.1.2.1.3 Unacknowledged mode

The messages are sent once and no feedback is sent from the receiver to the sender.
4.1.2.2 Radio Link Control (RLC)

The RLC protocol gives a transport service for the signaling entities Association Control Function (ACF), Radio Resource Control function (RRC), and the DLC User Connection Control function (DUCC). These three entities are used for signaling control messages between the AP and the MT [6].

4.1.2.2.1 Association Control Function (ACF)

The ACF is used when a MT wants to communicate with an AP. The MT listens to the Broadcast Channel (BCH) of different APs and selects the AP with the best radio link quality. The MT keeps on listening to the RLC Broadcast Channel (RBCH) to determine if the Network Operator -ID and Convergence Layer (CL) of the AP is acceptable. If the MT decides to continue the association, it requests and will be given a MAC-ID from the AP. More information about Channels will be given later in chapter 4.1.2.3.1.

After the MAC-ID is received the MT send its link capability to the AP containing information about

- Supported PHY modes
- Supported Convergence Layer
- Supported authentication and encryption procedures

The AP responds with supported PHY modes and a selected authentication and encryption procedure [6, 7].

The MT can disassociate either explicitly or implicitly. If the disassociation is explicitly, the MT notifies the AP that it no longer wants to communicate with the HiperLAN/2-network. Implicitly means that the MT has not sent a “MT alive”-message (see below) within a certain time period. All resources for allocated for the MT will be released [6].

During the association of a terminal the Dedicated Control Channel (DCCH) is established without any explicit signalling. The DCCH can be used for both downlink (from AP to MT) and uplink (from MT to AP) by the terminal [7].

4.1.2.2.2 DLC User Connection Control (DUCC)

To transfer user data (application data) between an AP and a MT a user connection has to be established. This connection is setup, maintained and closed by the DUCC. No user data transfers can be done before the connection is setup. A MT can have several simultaneous connections, each with its own priority.

4.1.2.2.3 Radio Resource Control (RRC)

The RRC is responsible for surveillance and efficient use of the frequency resources. These functions are controlled by the RRC:

- Handover
- Dynamic Frequency Selection (DFS)
- MT Alive
- Power saving

The RRC also supports other functions like Transmit Power Control and MT absence. For more information see [7].
4.1.2.3.1 Handover
The MT measures the signal strength from the surrounding APs. When it finds an AP with better radio link quality the MT request a handover. The handover can be done either by reassociation or by handover via the fixed network.

Reassociation means that the MT starts over with an association. The other alternative means that the new AP will retrieve information about the MT’s current connections and capabilities from the old AP via the fixed network. This reduces the loss of user plane data [6].

4.1.2.3.2 Dynamic Frequency Selection (DFS)
To avoid interference from neighboring HiperLAN/2 networks and other devices in the frequency band the DFS is used. The DFS in an AP choose a frequency based on measurements made by the AP or it is associated MTs. A AP may request a MT to measure and report the result back to the AP or the MT can also do the measure by itself and report to the AP [7].

4.1.2.3.3 MT Alive
To check if an MT and an AP can communicate with each other the MT alive function is used. The AP sends a “mt-alive-req” message to the MT. The MT responds with an “ack” message and continue sending “mt-alive” messages within a given time-interval. If the AP does not receive an “mt-alive” message it will send another mt-alive-req to the MT. If the MT does not respond with an “ack” message it will be disassociated from the AP [7].

4.1.2.3.4 Power saving
To reduce the power consumption an MT can join one of 16 different sleeping groups. When a MT has joined a sleeping group (i.e. the MT is in sleeping mode) it listens to the Broadcast Control Channel (BCCH) periodically instead of in every frame. The periodicity is given by:

\[2^n \text{ with } (1 \leq n \leq 16)\]

with the unit frame

The AP coordinates the sleeping groups such that the periodicity for all sleep groups coincide with sleep group where n=1. See MT1 in fig. 21. For MT2 n=2 and for MT3 n=3.

```
  MT3          MT1          MT2
  MT2          MT1          MT1
  MT1          MT1          MT1
```

Figure 21: Different sleep groups

When the sleeping period is over (i.e. the MT listens to the BCCH) the sleep state can be set to active either by the MT or by the AP, or the MT can continue in sleep mode [7].

4.1.2.3 Medium Access Control layer (MAC)
The MAC-protocol is based on a TDMA/TDD (Time Division Multiple Access / Time Division Duplex) scheme controlled from the AP, which means that the AP controls when a MT should receive data and when it is allowed to transmit data. Several MTs can transmit and receive data within a MAC-frame and the resources are dynamically controlled by the AP. The MAC-frame is always 2 ms long.
Each MAC entity (e.g., MT) in a radio cell gets a unique 8-bit MAC ID from the AP/CC during association. This ID is used when data and control information is send between MT and AP. It is also used to identify the MT in broadcast and multicast services.

The MAC-frame consists of a broadcast phase, a downlink phase, an uplink phase and a random access phase. The direct link phase in figure 22 is optional.

During the broadcast phase the AP sends information to all MTs about the structure of the rest of the MAC-frame, e.g., information about which and when a MT should receive and transmit data and the coding rate used.

In the downlink phase data is sent from the AP to the MTs. A MT receives the data during the time interval specified in the broadcast phase.

In the uplink phase data is sent from a MT to the AP.

In the optional direct link phase data is sent directly from one MT to another.

If a MT has not been given time in the uplink phase it can try to use the random access phase to send requests and information to the AP. If several MTs send requests at the same time it is possible that they collide and the AP never receives either of them. To inform the MT that the request was received properly the AP sends an acknowledge message during the broadcast phase of the next MAC-frame.

4.1.2.3.1 Logical and Transport channels

When the DLC-layer in an AP or MT wants to send a message (i.e., transport data) to the DLC-layer in another MT or AP, it constructs the message according to a predefined format. The format for a specific kind of message is always the same but the content changes. These messages are called logical channels and are abbreviated with 4 letters. An example of a logical channel is given in fig 23. The figure is only an example so the fields are not explained here. See [7] for further information.
Figure 23: Example of a logical channel (the Broadcast Control Channel – BCCH)

The logical channels are mapped on transport channels, i.e. a transport channel contains a logical channel. There are six different transport channels (abbreviated with 3 letters), which appear in different phases of the MAC-frame as shown in figure 24.

Figure 24: The transport channels in a MAC-frame. Broadcast Channel (BCH), Frame Channel (FCH), Access feedback Channel (ACH), Short and Long transport Channels (SCH and LCH) and Random Channel (RCH).

Fig. 25 to 27 shows in which transport channels the logical channels can be sent during downlink, uplink and direct link. For example, the RLC Broadcast Channel (RBCH) can be sent in a SCH or LCH during uplink. A SCH in downlink can contain one of the logical channels Link Control Channel (LCCH), RBCH and Dedicated Control Channel (DCCH).

The three first transport channels are only sent in downlink

Figure 25: Mapping between logical and transport channels for downlink
4.1.2.3.2 The MAC-frame

The time unit of a MAC-frame is called slot and is 400ns long, which means that a MAC-frame has 5000 slots. When referring to a specific place within the MAC-frame a 12-or 13-bit pointer is used to identify the slot. The beginning of the preamble for the first BCH is used as reference.

The 6 different transport channels that are used to construct the MAC-frame are:

<table>
<thead>
<tr>
<th>Transport channel</th>
<th>Direction</th>
<th>PHY mode</th>
<th>Length [octets]</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCH</td>
<td>Downlink (Broadcast)</td>
<td>Binary PSK and code rate ½.</td>
<td>15</td>
<td>Sent in every MAC frame for each sector.</td>
</tr>
<tr>
<td>FCH</td>
<td>Downlink (Broadcast)</td>
<td>Binary PSK and code rate ½.</td>
<td>Multiple of 27</td>
<td>Sent in every MAC frame for each sector that contains scheduled data.</td>
</tr>
<tr>
<td>ACH</td>
<td>Downlink (Broadcast)</td>
<td>Binary PSK and code rate ½.</td>
<td>9</td>
<td>Sent in every MAC frame for each sector.</td>
</tr>
<tr>
<td>SCH</td>
<td>DL/UL/DiL</td>
<td>Set in FCCH.</td>
<td>9</td>
<td>PHY mode is set and adapted per connection.</td>
</tr>
<tr>
<td>LCH</td>
<td>DL/UL/DiL</td>
<td>Set in FCCH.</td>
<td>54</td>
<td>PHY mode is set and adapted per connection.</td>
</tr>
<tr>
<td>RCH</td>
<td>Uplink</td>
<td>Binary PSK and code rate ½.</td>
<td>9</td>
<td>Contention based access.</td>
</tr>
</tbody>
</table>

Table 10: Transport Channels in a MAC-frame

4.1.2.3.2.1 Broadcast Channel (BCH)

The BCH is sent at the beginning of each MAC frame using BPSK and coding rate ½. An AP can have several sectorized antennas to improve the signal-to-noise ratio. The best sector is chosen for each MT to enhance the performance. If the AP has several sectors, one BCH is sent for each sector beginning with sector ID equal to 0 and increasing. The BCH is only transmitted once per MAC frame expect when the AP performs DFS related measurements.
The BCH is carrying a Broadcast Control Channel (BCCH) containing information about radio cell information such as network ID, AP ID and transmit power. It also contains pointers to the FCH and to the RCH indicating their location within the MAC frame.

4.1.2.3.2.2 Frame Channel (FCH)
If there is only one sector the FCH shall follow directly after the BCH. Otherwise a pointer in the BCCH is used to identify the starting point of the FCH.

The FCH carries the Frame Control Channel (FCCH). Blocks of Information Elements (IE) and CRC-blocks build the FCH. The structure is shown in fig. 28. There shall always be three IE elements, each with a length of 8 octets followed by a 24-bit (3 octets) CRC.

The IEs are always arranged in groups of three. If the number of IEs not is a multiple of three, padding elements are used to fill the FCH.

The IEs contain information about all the downlink, direct link and uplink transmission made in the MAC frame. They are also used to tell where in the frame empty parts are allocated. The empty parts are only used when a MT has been ordered to make DFS measurements.

4.1.2.3.2.3 Access feedback Channel (ACH)
The ACH carries the Random access Feedback Channel (RFCH). The RFCH contains feedback to the use of RCHs in the previous frame. If a MT sent a request to the AP during MAC-frame \( i \), it can read the response in the RFCH in MAC-frame \( i+1 \).

4.1.2.3.2.4 Short transport Channel (SCH) and Long transport Channel (LCH)
The SCH and LCH are used during downlink, uplink and direct link. The logical channels are mapped on these transport channels according to fig. 25-27. They are 9 respectively 54 octets long and the PHY mode for these channels are set in the FCH.
Table 11 shows the logical channels mapped to SCHs and LCHs.

<table>
<thead>
<tr>
<th>Logical Channel</th>
<th>Mapped to transport channel</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link Control Channel (LCCH)</td>
<td>SCH</td>
<td>ARQ-feedback messages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Discard messages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Resource Requests (only sent in uplink phase)</td>
</tr>
<tr>
<td>RLC Broadcast Channel (RBCH)</td>
<td>SCH or LCH</td>
<td>Broadcast RLC messages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Encryption seed</td>
</tr>
<tr>
<td>Dedicated Control Channel (DCCH)</td>
<td>SCH or LCH</td>
<td>RLC messages</td>
</tr>
<tr>
<td>User Data Channel (UDCH)</td>
<td>LCH</td>
<td>User data between the AP and an MT</td>
</tr>
<tr>
<td>User Broadcast Channel (UBCH)</td>
<td>LCH</td>
<td>User broadcast data</td>
</tr>
<tr>
<td>User Multicast Channel (UMCH)</td>
<td>LCH</td>
<td>User multicast data</td>
</tr>
</tbody>
</table>

Table 11: Different logical channels mapped to transport channels

**4.1.2.3.2.5 RCH**

If a MT wants to send control information to the AP and doesn’t have any granted SCH available, it uses the RCH. The RCH can carry LCCH (if it is a Resource Request), Association control Channel (ASCH) and Dedicated Control Channel (DCCH) data.

The ASCH conveys association requests and handover requests.

**4.1.3 Convergence layer**

The task of CL is to adapt service requests from higher layers to the services offered by the DLC. A special CL is designed for each supported core network. There are currently two types of CL, a cell based to support networks like ATM and UMTS and a packet based for Ethernet, IP and IEEE 1394 (Firewire) [7].

![Figure 30: HiperLAN/2 Convergence layer](image)

**4.1.3.1 Ethernet Convergence layer**

To support different technologies the packet based CL is divided in 2 main parts, a common part and a Service Specific Convergence Sublayer (SSCS), see fig 31.
The common part is divided in 2 sublayers, Segmentation And Reassembly (SAR) and Common Part Convergence Sublayer (CPCS).

The following sections give a description of how an Ethernet package is formatted and sent in Hiperlan/2.

### 4.1.3.1.1 Service Specific Convergence Sublayer (SSCS)

The Ethernet SSCS receives an IEEE 802.3 MAC-frame from the Ethernet layer. The frame consists of preamble, Start Frame Delimiter (SFD), destination address, source address, type and length field, payload and a frame check sequence (FCS). The preamble, SFD and FCS are discarded by the SSCS. For more information about these fields, see [9].

The destination field contains the 48-bit IEEE MAC address for the receiver and the source field the address of the sender. The type/length field gives the size of the payload or information about the Ethernet-MAC used. See [9] for more info.
4.1.3.1.2 Common Part Convergence Sublayer (CPCS)
The CPCS adds a padding field and a trailer field to the payload received from the SSCS. The trailer field contains the length of CPCS-payload and the padding field is added to make the total length a multiple of 48 octets.

![Figure 32: Mapping of a user data package](image)

4.1.3.1.3 Segmentation And Reassembly (SAR)
The SAR divides the package received from the CPCS in segments of 48 octets. It also adds a tag field, which is not used in the packet based convergence layer. All bits in this field are set to 0. The flag field only uses one of the bits and it indicates if it is the last segment.

4.1.3.1.4 DLC
The DLC adds a type field, a sequence number and a CRC to the payload. The type field shows what kind of message it contains (a user data channel, UDCH, in this case). The sequence number is added to ensure that the DLC delivers the payload to the SAR in correct sequence. The CRC field checks if the payload has been transferred correctly. These parts are the content of a UDCH, which is mapped to a LCH and sent by the physical layer.
Figure 33: Example for the transmission order of DLC messages
5 Quality of Service (QoS)

Quality of Service (shorthand QoS) corresponds to the goodness (quality) a certain operation (service) is performed with. Certain services (like multimedia applications or a simple phone call) need guarantees about accuracy, dependability and speed of the transmission performed. In common networks (TCP/IP based protocol suites) no special attention was paid to delay or throughput guarantees. With different requirements in real-time and multimedia applications, the networks must deal with an extendable set of QoS parameters. The most important QoS parameters are:

- Latency – the time between the message is sent and the receipt of the message. This time should be minimized.
- Jitter – occurs when video or voice transmitted does not arrive to the receiver in correct order or if the latency varies too much.
- Bandwidth – the amount of data that can be transmitted. QoS should provide the consistent data throughput capacity.

There are two types of QoS designed to meet these requirements, reservation based and priority based QoS.

In reservation based, the sender wants to reserve bandwidth for the transmission. The bandwidth is reserved all the way from the sender, through routers to the receiver.

A simpler way to achieve QoS is by giving the data packets different priorities. A packet with high priority is sent before a packet with lower priority. IP has eight levels of prioritisation for messages.

Ethernet also uses 8 levels to set the priority of a message, which makes it easy to map Ethernet traffic to IP-networks. The levels are specified in table 8 [10].

<table>
<thead>
<tr>
<th>User priority</th>
<th>Traffic Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(Default) Best Effort (BE)</td>
<td>Default LAN traffic</td>
</tr>
<tr>
<td>1</td>
<td>Background (BK)</td>
<td>Bulk transfers and other activities that are permitted on the network but that should not impact the use of the network by other users and applications</td>
</tr>
<tr>
<td>2</td>
<td>Spare</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Excellent Effort (EE)</td>
<td>For valued customers</td>
</tr>
<tr>
<td>4</td>
<td>Controlled Load (CL)</td>
<td>Traffic will have to conform to some form of Higher Layer admission control</td>
</tr>
<tr>
<td>5</td>
<td>&quot;Video&quot; (VI)</td>
<td>Characterized by less than 100 ms delay</td>
</tr>
<tr>
<td>6</td>
<td>&quot;Voice&quot; (VO)</td>
<td>Characterized by less than 10 ms delay</td>
</tr>
<tr>
<td>7</td>
<td>Network Control (NC)</td>
<td>Characterized by a &quot;must get there&quot; requirement to maintain and support the net-work infrastructure.</td>
</tr>
</tbody>
</table>

Table 12: Ethernet priority-levels

5.1 802.11a

In 802.11a there are two different access methods specified, asynchronous and synchronous.

The asynchronous, Distributed Coordination Function (DCF), is mandatory and it uses Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) to control if the medium is idle. It uses four different timeframes called Interframe Space (IFS) to determine who should gain access to the medium.

The other method is optional and it called Point Coordination Function (PCF). It uses a Point Coordinator (PC) to determine which STA that should transceive. The PC should be located in the AP and poll the STAs to send. When the PC is active it is called the Contention Free
Period (CFP) and it lasts until the PC releases the medium. The DCF can not access the medium at this time since it need the medium to be idle for a DCF IFS (DIFS) which is longer than the Short IFS (SIFS) that the PC needs to wait.

IEEE 802.11e is a working group with the purpose: “Enhance the current 802.11 MAC to expand support for applications with Quality of Service requirements, and in the capabilities and efficiency of the protocol”. There are many different proposals but there are three main ideas called Enhanced DCF (EDCF), Enhanced PCF (EPCF) and Hybrid Coordination Function (HCF).

### 5.1.1 DCF

The DCF is implemented in all stations using 802.11a. It uses CSMA/CA since it is impossible for a transmitting station to detect collisions in a wireless media. Therefore it senses the medium and wait until it has been idle for a DIFS before it can transmit.

The sensing can be done using Request To Send (RTS) and Clear To Send (CTS) that includes a duration number that is a indication on how long the medium will be occupied.

The STA sends a RTS to the AP to gain access to the medium. If the medium is idle the duration number is put in the Network Allocation Vector (NAV) of the other STAs and CTS is returned. The NAV is only updated if the duration number is larger than the present NAV value.

If an STA has a nonzero value in the NAV it is not allowed to access the medium. The NAV is then decremented so it is zero when the medium is idle again.

In cases when short data frames should be transmitted it can be very inefficient to use the RTS/CTS frames. The dot11RTSThreshold variable allows STAs to be configured to use RTS/CTS either always, never or on frames longer than a by the user specified length.[8]

#### 5.1.1.1 Interframe Space (IFS)

There are four different timeframes used to access the medium: Short, PCF, DCF and Extended IFS, see figure 34. [8]

![Figure 34: Different InterFrame Spaces](image)

**5.1.1.1 Short IFS (SIFS)**

SIFS is the shortest time gap and it is used for acknowledge and CTS frames, a subsequent MPDU of a fragment burst and by a STA responding to a polling by the PCF.
Since it is the smallest gap it is used by STAs to maintain access to the medium. It has the highest priority since other frames have to wait for a longer gap (PIFS, DIFS or EIFS) before they can access the medium.

5.1.1.1.2 PCF IFS (PIFS)
PIFS should only be used by STAs operating under the PCF to gain access to the medium at the start of the Contention Free Period (CFP).

5.1.1.1.3 DCF IFS (DIFS)
The DIFS is used by STAs operating under the DCF to transmit data and management frames. This is the time the medium must be idle after a successful transmission before a STA operating under the DCF can access the medium again.

5.1.1.1.4 Extended IFS (EIFS)
This is the longest IFS and it is used by STAs operating under the DCF when the last transmitted frame was erroneous. It is defined to provide enough time for another STA to acknowledge what was an incorrectly received frame before this STA starts transmission. If an error-free frame is received during the EIFS the STA resynchronises to the actual busy/idle state of the medium, so the EIFS is terminated and normal medium access is applied.[8]

5.1.1.2 Random Backoff Time
If a STA detects that the medium has been idle for a DIFS or an EIFS it should generate a random backoff time, unless the backoff timer already contains a nonzero value. This is done to prevent collisions since all STAs using DCF detects that the medium is idle. The backoff time is obtained from:

\[
\text{Backoff Time} = \text{Random()} \times \text{aSlotTime}
\]

Where aSlotTime is 9 µs and Random() is a function that returns a random number in the interval [0,CW]. CW is an abbreviation for Contention Window and it is a number in the interval [aCWmin, aCWmax] that in 802.11a is [15,1023]. The CW starts at aCWmin and is increased to a new value every time a STA must retransmit a frame. This is done until it reaches aCWmax where it remains until it is reset. The set of CW values shall be sequentially ascending integer powers of 2, minus 1 see picture 35. CW is reset to aCWmin after every successful attempt to transmit an MSDU or MMPDU. [8]
When the medium is idle the Backoff timer is decremented every aSlotTime, in this case 9 μs. HiperLAN. As soon the medium is busy the Backoff time is frozen. When the timer reaches zero has the STA access to the medium.

5.1.1.3 Transfer procedure
When a STA has detected that the medium has been idle for longer than a DIFS or an EIFS and it has finished its backoff procedure it is allowed to send.

A backoff procedure should always be started after a transmission where the More Fragments bit is set to zero. If it is set to one there are more fragments of the same MMPDU or MSDU left to send and the same STA accesses the medium after a SIFS. Fragmentation is used to
save bandwidth when errors occur. If the frame to be transmitted is large (maximum 4095 bytes) than it takes much time to retransmit the whole frame. With fragmentation you only have to retransmit the fragment of the frame that contains the error. Each fragment contains enough information to allow the complete MMPDU or MSDU to be reassembled from constituent fragments.

5.1.2 PCF
PCF uses a Point Coordinator (PC) to access the medium. The PC is located in the AP and therefore is PCF only usable on infrastructure network configurations. It is determined by the PC which STA that is allowed to send. The PC performs the role of a polling master and polls the STA that is allowed to send. It gives STAs Transmission opportunities (Tx-ops).

The PC uses Beacon management frames to gain control of the medium by setting the NAV in stations. Since PCF uses shorter IFS’s than DCF it always gain control of the medium. The PC keeps track of the transmissions and keeps it free from contention. [12]

5.1.3 Enhanced Cooperation Functions
IEEE 802.11e has currently not yet decided what proposal they will follow to enhance the QoS. Therefore follows a description of the functions that are most discussed right now.

5.1.3.1 EDCF
The EDCF is in general a DCF that can use different queues for different prioritised frames using Traffic Categories (TCs). An EDCF STA (ESTA) will have at most 8 prioritised queues, one for each TC. An EDCF AP (EAP) should have at least 4 physical queues. Every queue competes to access the media using an Arbitration IFS (AIFS) that is different for different TCs.

When the medium has been idle for an AIFS[TC] then that queue can transceive if their Random Backoff Time is zero. The different queues can have different Backoff Times and Contention Window. [12]

There is a sort of EDCF called p-DCF that uses carrier sense multiple access with adaptive contention (CSMA/AC) instead of carrier sense multiple access with collision avoidance (CSMA/CA). Under CSMA/AC the ESTA is using a station-based permission probability (PP) to obtain the Traffic Category Permission Probability (TCPP). There are 8 different TCPPs in correspondence with the priorities. This is managed using a Hybrid Coordinator that implements functions for both contention free access and enhanced contention access. If one network uses coordinated contention each TC is assigned a TCPP and is conceptually sent with that probability.

p-DCF only uses one Backoff Timer per station in distributed contention mode and it is updated using the TCPP. A station gets access to the medium if their PP is lower than a pseudorandom number X and its backoff timer is zero. If X is larger than the stations PP then the Backoff Counter (BC) is incremented by 1. BC is then used to set the backoff timer. The backoff timer is decremented when the ESTA senses the medium to be idle. AN ESTA using p-DCF uses the same rules to determine when the medium is idle, after DIFS with a correctly sent frame and after an EIFS if the frame was erroneous.
With distributed contention the TCPP only updates if it is zero or when a frame is transmitted from that TC. After a transmission TCPP is updated with respect to the priority of the TC and to if the transmission was successful. [13, 14]

5.1.3.2 EPCF

The Enhanced PCF uses 8 different priorities just like EDCF but it also uses parameters that allow 8 priorities per connection. The source/destination pairs of MAC-addresses, and the priority identify a connection. Each connection is mapped to a traffic specification that instructs the MAC how to best schedule those MSDU’s and minimise jitter and latency. [15]

5.1.3.3 HCF

The Hybrid Coordination Function combines EDCF with generalised PCF features. With this combination is the HCF able to work in the Contention Free Period (CFP) and the Contention Period (CP). HCF is also able to generate Contention Free Bursts (CFBs) in the CP to meet traffic specifications. A CFB is a “mini-CFP” in the CP with a max-duration that is limited to allow EDCF contention opportunities. To start a CFB shall the HC use EDCF access rules, i.e. after the applicable AIFS. If no traffic is queued then a low priority AIFS and CWmin should be used. The HC can start a CFB direct after a Data/Mgmt frame exchange from the same BSS if NF = 0, see figure 37. As long as the CFB duration is less than CFB limit can the HC exchange as many frames it desires. [16, 17]

![Figure 37: Use of CFB to access the medium](image)

5.2 HiperLAN/2

HiperLAN/2 uses a TDMA/TDD scheme controlled by the AP to schedule transmissions between MTs and APs. This makes it easier to implement QoS since the AP has information about all current transmissions and their QoS demands. The AP can allocate the consistent bandwidth for each connection to ensure QoS.
5.2.1 Resource reservation
The MT can negotiate that the AP shall schedule a number of LCHs and SCHs for a certain number of MAC-frame. This is done during a connection setup. When the MT has been granted the fixed capacity and is ready to start using it, a Resource Request is sent to the AP and the transmission begins in the next frame.

5.2.2 Priority in Ethernet Convergence Layer
During association the AP allocates between 1 and 8 connections for the MT. Each connection has a queue where data that is going to be transmitted is put. When data is received from the Ethernet layer it is put in one of the queues. The priority and the number of queues determine which queue to use. Each queue is identified by the DLC Connection ID (DLCC-ID). The mapping between traffic types and DLCC-Ids is showed in table 13.

<table>
<thead>
<tr>
<th>Number of DLCCs (connections)</th>
<th>Traffic Classes</th>
<th>DLCC-ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{ Best Effort, Excellent Effort, Background, Voice, Controlled Load, Video, Network Control }</td>
<td>Lowest</td>
</tr>
<tr>
<td>2</td>
<td>{ Best Effort, Excellent Effort, Background }</td>
<td>Lowest</td>
</tr>
<tr>
<td></td>
<td>{ Voice, Controlled Load, Video, Network Control }</td>
<td>2\textsuperscript{nd} lowest</td>
</tr>
<tr>
<td>3</td>
<td>{ Best Effort, Excellent Effort, Background }</td>
<td>Lowest</td>
</tr>
<tr>
<td></td>
<td>{ Controlled Load, Video }</td>
<td>3\textsuperscript{rd} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Voice, Network Control }</td>
<td>2\textsuperscript{nd} lowest</td>
</tr>
<tr>
<td>4</td>
<td>{ Background }</td>
<td>4\textsuperscript{th} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Best Effort, Excellent Effort }</td>
<td>Lowest</td>
</tr>
<tr>
<td></td>
<td>{ Controlled Load, Video }</td>
<td>3\textsuperscript{rd} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Voice, Network Control }</td>
<td>2\textsuperscript{nd} lowest</td>
</tr>
<tr>
<td>5</td>
<td>{ Background }</td>
<td>4\textsuperscript{th} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Best Effort, Excellent Effort }</td>
<td>Lowest</td>
</tr>
<tr>
<td></td>
<td>{ Controlled Load, Video }</td>
<td>5\textsuperscript{th} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Voice, Network Control }</td>
<td>2\textsuperscript{nd} lowest</td>
</tr>
<tr>
<td>6</td>
<td>{ Background }</td>
<td>4\textsuperscript{th} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Best Effort }</td>
<td>Lowest</td>
</tr>
<tr>
<td></td>
<td>{ Excellent Effort }</td>
<td>6\textsuperscript{th} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Controlled Load }</td>
<td>3\textsuperscript{rd} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Video }</td>
<td>5\textsuperscript{th} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Voice, Network Control }</td>
<td>2\textsuperscript{nd} lowest</td>
</tr>
<tr>
<td>7</td>
<td>{ Background }</td>
<td>4\textsuperscript{th} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Best Effort }</td>
<td>Lowest</td>
</tr>
<tr>
<td></td>
<td>{ Excellent Effort }</td>
<td>6\textsuperscript{th} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Controlled Load }</td>
<td>3\textsuperscript{rd} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Video }</td>
<td>5\textsuperscript{th} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Voice }</td>
<td>2\textsuperscript{nd} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Network Control }</td>
<td>7\textsuperscript{th} lowest</td>
</tr>
<tr>
<td>8</td>
<td>{ Background }</td>
<td>4\textsuperscript{th} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Best Effort }</td>
<td>Lowest</td>
</tr>
<tr>
<td></td>
<td>{ Excellent Effort }</td>
<td>6\textsuperscript{th} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Controlled Load }</td>
<td>3\textsuperscript{rd} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Video }</td>
<td>5\textsuperscript{th} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Voice }</td>
<td>2\textsuperscript{nd} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Network Control }</td>
<td>7\textsuperscript{th} lowest</td>
</tr>
<tr>
<td></td>
<td>{ Spare }</td>
<td>8\textsuperscript{th} lowest</td>
</tr>
</tbody>
</table>
Excellent Effort and Background traffic will be put in the queue belonging to the connection with DLCC-ID 4.

Fig 38 shows all traffic types and in which queue they are put. The Ethernet priority for each traffic type is given in the brackets. See table 12 for more information about the Ethernet priorities.

![Diagram showing Ethernet packages mapping to connection queues.](image)

Figure 38: Example of mapping Ethernet packages to connection queues.
6 Security

One of the main issues in wireless communication is how to provide enough security. A wireless LAN has unique security problems compared to an ordinary LAN. To monitor an ordinary LAN one have to be physically near or in direct contact with the wire medium. With a wireless LAN it is enough for the eavesdropper to be inside the broadcast range.

Since it is impossible to control the broadcast completely it is necessary to encrypt the messages to be sent in a way that prevents interception. To encrypt a message one needs the message, the encryption key and the encryption algorithm and obviously to decrypt it one needs the encrypted message, the encryption key and the encryption algorithm.

An eavesdropper in a WLAN has the encrypted message and the algorithm so it is very important to protect the key.

The encryption of the message differs a bit between 802.11a and HiperLAN/2 so first is a list of security events followed by how the two different standards solves the problems.

6.1 Security events

There can occur many different security events in a WLAN and here they are listed in five different categories.

6.1.1 Human initiated events

Most of the security events are a result of a human error. Two common errors are improper system configuration and system software errors. This sort of errors cannot be avoided by better specifications and encryption.

6.1.2 Data privacy

Data privacy is when an unauthorized user gains access to the unencrypted traffic between two points. To decrypt the message the user must have access to the encrypted message, the algorithm and the key. The eavesdropper receives the encrypted message and the algorithm is given in the LAN specification. The key can be discovered if the eavesdropper have access to some of the unencrypted plain text assumed no additional noise or filler characters are added to the encrypted text.

Most encryption algorithms use:

\[ E(i) = k(i) \text{<operator>} P(i) \text{ for } i = 1 \text{ to number of bits in the key} \]

Where

- \( E(i) \) is the i-th bit of the Encrypted message
- \( K(i) \) is the i-th bit of the key
- \(<\text{operator}>\) is the mathematical operator
- \( p(i) \) is the i-th bit of the plain text operator

To increase security it is possible to add non-repetitive elements to the encrypted message:

\[ E(i) = k(i) \text{<operator>} P(i) \text{<operator>} m(i) \text{ for } i = 1 \text{ to number of bits in the key} \]

Where

- \( m(i) \) is the i-th bit of the message sequence key
It is still relatively easy to obtain the key if you have the first part of two encrypted messages, the first part of two plain messages and the relationship between sequence numbers.

If the encryption algorithm is changed so it randomly adds bits to the plain text message the equation always have a random pattern.

One can add additional complexity if the network layer always sends maximum IP block sizes. That implies that smaller messages get filled up with filler bits. By randomly adding byte sequences and use fixed block size makes the intruder unable to reliably guess when a plain text is transmitted.

This approach to encryption demands for a pseudo key to be created for every block of n characters. The key should be based on the previous encryption results.

6.1.3 Data forgery
A data forgery occurs when an unauthorized user inserts data to the network as a valid user. There are two different ways that valid data can enter the network, replay and mimicking.

6.1.3.1 Replay
The method of replay is very straightforward; you simply record the traffic and then try to replay it later. The replay can be either an exact copy or slightly modified. The hacker needs a limited knowledge of the plain text message, some knowledge of the message format, and good method to replay a message to create a modified message.

It is very useful to start a replay attack when a privileged user grants temporary rights on a user’s station. When the intruder then sends a replay the terminal grants the intruder temporary rights.

One solution to the replay problem is to assign every message a message number and have a fixed relationship between the numbers. A replay attack would then send messages with incorrect message number that would be rejected. But since the message number cannot go to infinity it has to loop through a certain set of numbers. That gives the intruder a little possibility to send with the proper message number. This issue can be avoided by letting the message number be a part of the encrypted message. This implies that when all the numbers are used, a new encryption key is required to send new messages.

6.1.3.2 Mimicking
Mimicking is used when the intruder has knowledge of the key and has total access to the network. Then the intruder mimics a valid station in the network and transmits data using the valid stations ID. Since the key is discovered it is impossible for the receiving station to distinguish the intruder from the valid user.

If the detection of data forgery is moved from the MAC-layer to the PHY-layer the RF-interface could use the subtle differences in different transmitters to identify a data forgery. This has been researched by the cellular industry and it should be possible to use their discoveries in WLANs.

6.1.4 Denial of service
This is a problem of flooding. An intruder can flood the network so the channel gets jammed and no other traffic can be transmitted. The messages that are sent can be either valid or
invalid. The intruder can also flood a receiving station until it has run out of battery power and accomplishes denial of service.

Like “Human initiated errors” this is one type of errors that better specifications cannot avoid.

6.1.5 Hardware errors

When a hardware error occurs it should be detected and prevented at the PHY layer. Examples of hardware errors are that the hardware could cease to function, jam the channel by sending constant, or send random patterns on the channel. Another station can receive this random pattern as a valid message. [18]

6.2 Solutions in 802.11a

802.11a uses the Wired Equivalent Privacy (WEP) algorithm to provide secure message sending. This algorithm is intended to provide equal security as a wired network.

The main goals for WEP is

- **Confidentiality**: The most important goal is to prevent eavesdropping
- **Access control**: To prevent strangers from accessing the network and steal bandwidth
- **Data integrity**: Prevent tampering of sent data

The confidentiality depends on an external key management service to distribute data eniciphering/dechiphering keys. The longer a key is the harder it is to find out its value. In standard WEP is the key 40-bit long. This length is a result of US Government export restrictions regarding technology that contains chryptography. The restrictions are no longer in effect so some vendors offers so called “128-bit” versions, despite it actually uses 104 bits. This extended keys makes a brute-force attack impossible with todays technology.

The enicipherment procedure is shown in figure 39.

![Figure 39: WEP encipherment block diagram](image-url)
Since the Secret key is the same during all transmissions it is important to concatenate it with the Initialization Vector (IV) to produce a Seed that changes with every transmission. This is done to prevent that two messages are encrypted with the same Key Sequence and by that simplify for an intruder.

The WEP Pseudo-Random Number Generator (PRNG) transforms the relatively short Seed to an arbitrary long Key Sequence. The generator uses the RC4 algorithm provided by RSA Data Security, Inc, to generate a Key sequence based on both IV and the key. This sequence is one of the inputs to the encryption procedure.

The plaintext is used to create an Integrity Check Value (ICV) that is used to guarantee that the received message equals the sent one. This is done using the CRC-32 algorithm, CRC is short for Cyclic Redundancy Code. The ICV is then concatenated with the plaintext forming the second input to the encryption.

The encryption is a XOR-operation on the two inputs that generates the ciphertext. The ciphertext forms the message together with the IV, figure 40.

\[
\begin{array}{|c|c|c|}
\hline
\text{IV} & \text{Data (PDU)} & \text{ICV} \\
\hline
4 & \geq 1 & 4 \\
\hline
\end{array}
\]

*Figure 40: Encrypted Frame Body, sizes in octets*

The IV has to be sent with every message since it differs from message to message. The IV is then used to decipher the message, figure 41.

\[
\text{Secret Key} \quad \text{IV} \quad \text{Ciphertext} \quad \text{Seed} \quad \text{WEP PRNG} \quad \text{ICV} \quad \text{Plaintext} \quad \text{ICV'} \quad \text{ICV' = ICV?}
\]

*Figure 41: Decipherment block diagram*

The process is now simply done backwards. The IV and the Secret Key forms the Key Sequence that together with the ciphertext is put into the XOR-gate. The result from the XOR is divided into the Plaintext and the ICV. From the Plaintext is a new Integrity Check Value calculated that is called ICV’. The final step is then to compare ICV with ICV’ to see if there are any errors in the message.
Despite encryption with WEP there are several ways to eavesdrop and find out what the secret key is. IEEE is therefore working on enhancements in security.

6.2.1 Enhancement proposals
The IEEE 802.11i task group has as goal to “Enhance the current 802.11 MAC to provide improvements in security”. They are striving to implement the Enhanced Security Network (ESN). There are several different elements in ESN so systems may wish to use only some of them in their solution.

The 802.11i task group decided at their July meeting 2001 that they would not specify any mandatory authentication protocol or key distribution mechanism. This means that security solutions can vary a lot between different vendors. Three different elements in ESN is listed below.

6.2.1.1 WEP2/WEP2+
WEP2 is as the name implies an improvement of the WEP-algorithm. The main improvements are:

- The IV is extended to 128 bits instead of 32. This makes unintentional reuse of the key sequence much less likely. But without IV replay protection is unintentional reuse still possible.
- The key may be changed periodically. This should be done using the IEEE 802.1X reauthentication process. This however requires new hardware.

WEP2+ offers keyed IV to prevent replaying and Message Integrity Code (MIC) for message authentication.

6.2.1.2 Advanced Encryption Standard (AES)
IEEE 802.11i has passed a motion to adopt the OCB algorithm in AES.

AES-OCB provides strong encryption, replay protection and message authentication. Instead of a IV it uses a Nonce that only needs to be unique during one keying session. One of OCB’s advantages is that it uses a small overhead.[19, 20]

6.2.1.3 Encapsulating Security Payload (ESP)
ESP is an existing well understood encrypting envelope that are very flexible in which encrypting and authentication transforms are supported with it. For example is several encryption methods with 64 bit IV and authentication methods using 96 bit represented.

ESP is well defined in RFC 2406. [21]
The format of the ESP is shown in figure 42.

The Sequence Number implies that the same key cannot be used with the IV, when the Sequence Number rolls over rekeying is forced.

There are different ways to create a new SPI for every connection. The most straightforward is to implement a Key Management Protocol (KMP) but it can also be implemented by using the STAs MAC-address and concatenate and hash it with some internal parameters.[19]

6.3 Solutions in HiperLAN/2

To provide confidentiality for data transmissions, an encryption algorithm is used. It is possible to encrypt UDCH, UMCH, UBCH and DCCH messages carried by LCHs. Figure 43 shows LCH encryption and decryption.

![Figure 35: Encryption and decryption of LCHs](attachment:image.png)
Each 54-octets LCH is encrypted as follows:

- The encryption module is loaded with a 64-bit (8-octet) Initialization Vector (IV).
- The keystream output from the encryption is XORed with the 8 first octets of the plaintext and out comes the ciphertext. The keystream is also fed back to the encryption module to generate a new keystream for the next 8 octets.
- The procedure continues until the last block of plaintext has been encrypted.

The decryption on the receiver side uses the same input and encryption function as the sender. Note that the receiver uses an encryption function and not a decryption function. The keystream is then XORed with the ciphertext to give the plaintext [7].

### 6.3.1 Initialization Vector (IV)

Each LCH transmitted uses an IV for encrypting/decrypting the first 8 octets. Both the AP and MT have an IV generator to generate a new IV for each LCH. As shown in previous chapters several LCHs can be transmitted from between an AP and a MT during one MAC-frame. The IV for the first LCH is generated using a seed generator in the AP and the 12 LSBs of the startpointer in Frame Control Channels (FCCH) Information Element (IE). See figure 44.

![Figure 44: IV generator initialization](image)

For each of the following LCHs for that particular connection a new IV is generated according to fig 45. Bit 4, 3, 1 and 0 are XORed to generate the next MSB and the original content is shifted one step to the right [7].

![Figure 45: IV generator](image)

### 6.3.1.1 Seed generation

The seed generator generates a new code for each MAC-frame. This is done according to fig. 46. Bit 3 and bit 0 are XORed to generate the new MSB and the original content is shifted one step to the right.

![Figure 46: Seed generator](image)
The initial value for the seed-generator in an AP is any random bit sequence, except all zeros. Once it is initialized the code is sent to all MTs. Sending the code is done every Nth frame where N is a number determined by the vendor. Both the AP and the MT has a seed-generator so the MT is able to generate the right code during frames when the AP does not send it[7].

6.3.1.2 The Startpointer
For each connection between the AP and MT, an information element (IE) is sent. The IE contains information about how many SCHs and LCHs to be transmitted and a pointer that says where in the MAC-frame the transmission begins. This is the pointer used to create the first IV for a LCH.

6.3.2 Encryption
HiperLAN/2 uses the Data Encryption Standard (DES) or TripleDES algorithms to encrypt messages. DES is mandatory for AP and MT while TripleDES is optional.

The DES algorithm uses 56-bit key and has no known weakness except some weak keys that should not be used. More information can be found in “Guidelines for Implementing and Using the NBS Data Encryption Standard” provided by the US National Bureau of Standards.

The key(s) is exchanged during the setup between the MT and AP.

In TripleDES three different keys are used. The keystream (IV for the first 8 octets) is first encrypted with key 1, decrypted with key 2 and then encrypted again with key 3. See fig 47[7].

![Image of TripleDES encryption process]

**Figure 47: TripleDES encryption**
7 Differences between HiperLAN/2 and 802.11a

There are some main differences between HiperLAN/2 and 802.11a that makes it hard to converge them and make one worldwide accepted standard.

The 5GHz Globalization and Harmonization Study Group is joint project between ETSI/BRAN, IEEE and MMAC. MMAC is a Japanese organization and MMAC is an abbreviation for Multimedia Mobile Access Communication system. The Study Group is called 5GSG and it has a global scope.

The goal is to find potential ways to drive the different standards towards convergence to a single global WLAN standard that results in satisfactory product all over the world.

Since the physical layers differ very little between the different standards are 5GSG and this chapter focusing on the differences in the DLC protocol.

7.1 Security

One very important difference between HiperLAN/2 and 802.11a is how the encryption key is created and used. In 802.11a is the key shared between all STAs but in HiperLAN/2 it is unique for every connection.

The use of same key is convenient for 802.11a since it uses a Distributed Coordination Function which means that every STA can send to any other STA without having to send it through the AP. On the other hand is this decreasing security a lot since the key never changes. An eavesdropper can obtain the key by listening to several different STAs.

Also the use of IV differs a little between the standards. In HiperLAN/2 is the IV only used for the first 8 octets in a LCH and the rest of the LCH encryption is done with the previous keystream and the key, see figure 43. When a new LCH is sent the IV is rightshifted according to figure 45. A new IV is created for every new connection since the startpointer changes.

In 802.11a is the same IV used for the whole transmission and it is changed when a new message is to be sent. The IV is sent unencrypted in the message so the receiving STA can decrypt the message.

802.11a and HiperLAN/2 also uses different encryption algorithms. HiperLAN/2 uses DES and TripleDES which have 56 respective 168 bits. The TripleDES-algorithm uses 3 different keys to encrypt a message. First is the first key used to encrypt the message, then is the second used to decrypt it and finally is it encrypted again with the third key.

802.11a uses the WEP-algorithm that has 40 bit keys. Some vendors offer 104 bit keys to provide better security. There are also many other proposals to improve security but 802.11a has decided not to make them mandatory.

7.2 MAC

There are big differences in the MAC-layer between 802.11a and HiperLAN/2.

The main difference is that 802.11a uses carrier-sense multiple access with collision avoidance (CSMA/CA) when transmitting and HiperLAN/2 uses time-divided multiple access with time division duplex (TDMA/TDD).
7.2.1 Access methods
HiperLAN/2 uses a central controller (the AP) to control all communication between APs and MTs. The MAC-frame is always 2 ms long and during this time all MTs connected will have a period for communication to and from the AP.

In 802.11a the MAC listens if there is any traffic on the channel. If it is free it transmits the data. Using CSMA/CA gives the MTs the possibility to communicate directly without an AP. If power is sensed on the channel the MT waits a random time before trying to access it again.

7.2.2 Frequency and antenna
In HiperLAN/2 systems the transmission channel can change dynamically. The AP makes regular measurements and chooses the channel with least interference. This feature is not implemented in 802.11a.

HiperLAN/2 also has the possibility to implement several sectorized antennas to reduce the transmit power and increase the transmission rate.

7.3 Quality of Service
When transmitting data over a network, it is sometimes desirable to give some data priority over other. In voice transmissions for example, it is important that the delay is very little. To ensure the quality for different transmissions Ethernet has eight traffic types with different priorities.

In an Ethernet implementation of HiperLAN/2 these traffic types are used to schedule data transmissions and ensure quality of service. A MT has several (up to eight) different queues with in which the data packet is queued. The traffic type determines which queue to select.

In HiperLAN/2 there is also a possibility to reserve a fixed bandwidth for data transmissions.

In 802.11a there are several proposals on how to ensure quality of service. One of them, EPCF, is very similar to HiperLAN/2 with a central controller and 8 priority queues. None of the proposals are final and the date when to decide which proposal that is going to be standard is not set.
8 Test program

The program we developed is a foundation for a program that would test a WLAN-card. The idea was that Bluetronics would use it to develop a test-program for an 802.11a-CardBus card, which was under development. Unfortunately the project was terminated and our part never used.

To understand this program well, good knowledge of how the PC system works, good knowledge in Win2000 and the CardBus architecture is needed. See [3] and msdn.microsoft.com for more information about these subjects.

The program works as follows:

1. A driver is loaded to give the application rights to directly read from and write to I/O ports. This is normally not allowed in Win2000.
2. Another driver is loaded to search the PCI-bus for CardBus-to-PCI bridges.
3. The CardBus Bridge Configuration, ExCA and CardBus socket registers are read and stored in variables.
4. The graphical interface is loaded to show what is stored in the registers.
5. The I/O port and memory addresses used are read from the ExCA-register. When we developed the application, we used a 16-bit PCMCIA-card. If a CardBus card would have been used the addresses are stored in the CardBus bridge registers.

Figure 48 shows the GUI.

![Figure 48: Our GUI](image)

A short description of the five tabs is given below.
8.1 Cardbus Bridge Configuration Registers
Amongst others the CardBus bridge configuration registers specifies:
- The I/O and memory space assigned to the bridge
- Device and vendor ID
- A pointer to the CardBus- and ExCA socket registers (explained below)

8.2 Exchangeable Card Architecture (ExCA) Registers
The ExCA registers are used only by 16-bit PC Cards. These registers stores which resources a card has been assigned (memory and I/O space) and provides interrupt control, power control, controller status and card status.

8.3 CardBus Socket Registers
The PC Card standard requires that a CardBus bridge implements four socket registers and one control register. When the card generates a status changed interrupt, PCMCIA software can determine the cause of the interrupt by reading the socket registers. They are also used to mask out events that normally would cause a status changed interrupt and for debugging. The control register is used to apply power to the socket interface.

8.4 I/O port space
When a WLAN-card is installed in a computer it is assigned port and memory resources by the Operating System (OS). The ports are used by the computer to communicate with the card. For example, internal registers can be accessed through the ports assigned to the card. Exactly how many ports and how they are used is up to the vendor to decide.

8.5 Memory space
The memory space assigned to a card contains the Card Information Structure (CIS). The CIS contains information about the cardtype, resource requirements and configuration options.
9 Conclusions
Here follows our thoughts and conclusions about HiperLAN/2 and 802.11a. We have focused on the four different areas we consider important.

9.1 Implementation
Since both standards uses the same signals in and out of the MAC/Baseband circuit, it should be possible to implement them on the same chip. There will be two different MAC-functions but after encryption PDUs are processed in the same way. This implies that all transmission rates for both HiperLAN/2 and 802.11a should be represented. For example the baseband could use the 802.11a baseband extended with the 27 Mbit/s transmission rate that exists in HiperLAN/2, see table 1.

The choice between the standards could be implemented in software and the user chooses which net to use. A hardware solution where the MAC makes the choice is harder to implement in a straightforward way.

9.2 Security
The security is now better in HiperLAN/2. A system that uses the same key during a long time can be hacked, it is just a matter of time and computer capacity. It takes several hours to hack 802.11a and even longer to hack HiperLAN/2, but if the key is changed several times per hour the security is not a problem in either of the systems. This can be done by implementing a key exchange function or use the Kerberos-function in windows 2000.

9.3 Quality of Service
Today has HiperLAN/2 a better QoS especially for video and sound transmissions. This is because they use the same priority queues as Ethernet. They can also use resource reservation to guarantee a certain transmission a certain rate. However, this feature is seldom used in computer networks today.

802.11a are working on some proposals but since nothing is decided is it hard to say which one of the two standard that is going to have the best QoS in the future. Probably will they offer similar functions and options.

9.4 Future
We believe that HiperLAN/2 must release products very soon if they are about to establish their standard. 802.11a has a better starting point since the 802.11b standard is the dominating standard for WLANs operating at 2.4GHz. This and the fact that 802.11a products already is available on the market makes it hard for HiperLAN/2 to be the leading standards for WLAN in the 5GHz band.

But on the other hand, the HiperLAN/2 standard is much more finished and worked through. It has better QoS and security and supports both sectorized antennas and dynamic frequency selection (DFS). In an article from wireless europe [www.wireless.iop.org] it is stated that wireless system in the 5GHz band must support DFS to be allowed to in Europe. If that is a fact IEEE must modify their standard if it wants 802.11a products on the Europe market.
10 References:

10.1 Books and publications

1. Jim Geier
   *Wireless LANs*
   ISBN 1-57870-081-7

2. Bob O’Hara, Al Petrick
   *802.11 Handbook, A Designers Companion*
   ISBN 0-7381-1855-9

3. Don Anderson, Tom Shanley
   *Cardbus System Architecture*
   Addison Wesley Developers Press, 1996
   ISBN 0-201-40997-6

4. Tony Fredriksson
   *IEEE 802.11a and HiperLAN/2 – Standards for Wireless LAN*
   Department for Applied Physics and Electronics, Umeå University, 2000
   Master of thesis work EL0047

10.2 Internet

5. Andreas Hettich, Matthias Schröther
   *IEEE 802.11 or ETSI BRAN HIPERLAN/2: Who will win the race for a high speed wireless LAN standard?*
   Communication Networks, Aachen University of Thechnology

   *HiperLAN/2 – The Broadband Radio Transmission Technology Operating in the 5 GHz Frequency Band*
   HiperLAN/2 Global Forum

7. *Broadcast Radio Access Network (BRAN); HiperLAN Type 2*
   European Telecommunication Standards Institute, 2000
   [http://www.etsi.org/technicalactiv/hiperlan/hiperlan2.htm](http://www.etsi.org/technicalactiv/hiperlan/hiperlan2.htm)

8. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications*
   ANSI/IEEE Std 802.11, 1999 and IEEE Std 802.11a, 1999
   [http://standards.ieee.org/getieee802/802.11.html](http://standards.ieee.org/getieee802/802.11.html)
9. *Logical Link Control*
   ANSI/IEEE Std 802.2, 1998
   [http://standards.ieee.org/getieee802/802.2.html](http://standards.ieee.org/getieee802/802.2.html)

10. *Common specifications--Media access control (MAC) Bridges*
    ANSI/IEEE Std 802.1D, 1998

11. Nikita Borisov, Ian Goldberg, and David Wagner.
    *Intercepting Mobile Communications: The Insecurity of 802.11.*

References 12-20 can all be found at:

12. Mathilde Benveniste, Greg Chesson, Wim Diepstraten, Maarten Hoeben, Aman Singla, Harold Teunissen, Menzo Wentik
    *E-DCF Proposed Draft Text, IEEE 802.11-01/131r1*

13. Jin-Meng Ho, Sid Schrum, Khaled Turki, Don Shaver, Matthew B. Shoemake
    *Proposed Text for p-DCF Contention Access Enhancement, IEEE 802.11-01/138r1*

14. Jin-Meng Ho, Sid Schrum, Khaled Turki, Don Shaver, Matthew B. Shoemake
    *Presentation for Proposed p-DCF Contention Access Enhancement, IEEE 802.11-01/139*

15. *QoS MAC Enhancements Joint Proposal, IEEE 802.11-00/071r1*

16. Michael Fischer
    *Hybrid Coordination Function (HCF) Frame Exchange and NAV Details, IEEE 802.11-01/109r2*

17. Wim Diepstraten
    *HCF medium access rules, IEEE 802.11-01/383r3*

18. Alan Chikinsky
    *Wireless LAN Security Threats, IEEE 802.11-01/258*

19. Robert Moskowitz
    *Using the Encapsulating Security Payload (ESP) for WLAN privacy and protection, IEEE 802.11-01/510r0*

20. Carlos Rios
    *Optional MAC-level Authentication and Encryption Key Management, IEEE 802.11-01/465r0*

21. S. Kent and R. Atkinson
    *IP Encapsulating Security Payload (ESP)*
11 List of figures and tables

11.1 Figures
Figure 36: The different components of a WLAN ............................................. 6
Figure 37: Overview of a WLAN transceiver ................................................. 7
Figure 38: Components of the radio ............................................................. 7
Figure 39: One octet is chosen to the example .............................................. 8
Figure 40: Creating of the output streams X and Y ........................................ 9
Figure 41: The octet from figure 4 is modulated and sent on 4 sub carriers ....... 10
Figure 42: Example of how OFDM saves bandwidth .................................. 10
Figure 43: The different layers in 802.11a and the LLC layer ..................... 11
Figure 44: The general frame format of a MPDU ....................................... 12
Figure 45: Frame control field ................................................................. 12
Figure 46: Request to send frame ............................................................. 15
Figure 47: Clear to send frame ................................................................. 16
Figure 48: Power save poll frame ............................................................. 16
Figure 49: CF-End and CF-End + CF-Ack frame ...................................... 16
Figure 50: Fields in the management frames ............................................. 17
Figure 51: Sub fields in the Capability information field ............................ 18
Figure 52: The parts of a PSDU ............................................................. 20
Figure 53: The Signal field ................................................................. 21
Figure 54: HiperLAN/2 protocol stack .................................................... 23
Figure 55: Reference configuration of transmitter .................................... 24
Figure 56: Different sleep groups .......................................................... 27
Figure 57: Phases in the MAC-frame ....................................................... 28
Figure 58: Example of a logical channel (the Broadcast Control Channel BCCH) 29
Figure 59: The transport channels in a MAC-frame. Broadcast Channel (BCH), Frame Channel (FCH), Access feedback Channel (ACH), Short and Long transport Channels (SCH and LCH) and Random Channel (RCH) ........................................ 29
Figure 60: Mapping between logical and transport channels for downlink ...... 29
Figure 61: Mapping between logical and transport channels for uplink ........ 30
Figure 62: Mapping between logical and transport channels for direct link ...... 30
Figure 63: Structure of a FCH ............................................................. 31
Figure 64: Responses to RCH use in ACH .............................................. 31
Figure 65: HiperLAN/2 Convergence layer .............................................. 32
Figure 66: Different parts of CL ........................................................... 33
Figure 67: Mapping of a user data package .............................................. 34
Figure 68: Example for the transmission order of DLC messages ............... 35
Figure 69: Different InterFrame Spaces ............................................... 37
Figure 35: Example of contention Windows ........................................... 39
Figure 36: Example of Backoff times and the use of DIFS ......................... 39
Figure 37: Use of CFB to access the medium ........................................... 41
Figure 38: Example of mapping Ethernet packages to connection queues ...... 43
Figure 39: WEP encipherment block diagram ....................................... 46
Figure 40: Encrypted Frame Body, sizes in octets ................................... 47
Figure 41: Dechiferment block diagram ............................................... 47
11.2 Tables

Table 1: Data rates and modulation techniques .......................................................... 8
Table 2: Different code rates and puncturing pattern .................................................. 9
Table 3: Frame types and subtypes ............................................................................. 13
Table 4: To/From DS combinations in data type frames .............................................. 13
Table 5: The meaning of different combinations of CF-Pollable and CF-Poll request in a STA ............................................................................................................. 18
Table 6: The meaning of different combinations of CF-Pollable and CF-Poll request in an AP ............................................................................................................. 18
Table 7: The reason why a Disassociation or Deauthentication is generated .............. 19
Table 8: Meaning of the status code .......................................................................... 19
Table 9 PHY mode for HIPERLAN/2 .......................................................................... 25
Table 10: Transport Channels in a MAC-frame .......................................................... 30
Table 11: Different logical channels mapped to transport channels ......................... 32
Table 12: Ethernet priority-levels .............................................................................. 36
Table 13: Mapping between traffic types and DLCC-IDs ......................................... 42
Appendix A, IFS Calculations

aSIFSTime and aSlotTime is defined by the PHY.

\[
aSIFSTime = 16 \mu s = aRxDelay + aRxPLCPDelay + aMACProcessingDelay + aRxTxTurnaroundTime
\]

aRxDelay = The nominal time (in µs) before PMD sends a PMD-DATA.indicate to the PLCP when the end of a symbol arrives at the air interface.

aRxPLCPDelay = The nominal time (in µs) it takes the PLCP layer to deliver a bit from the PMD receive path to the MAC.

aMACProcessingDelay = The nominal time (in µs) it takes the MAC layer to process a frame and prepare a response to the frame.

aRxTxTurnaroundTime = The maximum time (in µs) that the PHY requires to change from receiving to transmitting.

\[
aSlotTime = 9 \mu s = aCCATime + aRxTxTurnaroundTime + aAirPropagationTime + aMACProcessingDelay.
\]

aCCATime = The minimum time (in µs) the CCA mechanism has available every time slot to assess the medium to determine whether it is busy or idle.

aAirPropagationTime = The predicted time (in µs) it takes a signal to go from the transmitting station to the receiving station.

The PIFS and DIFS are derived by the following equations:

\[
PIFS = 25 \mu s = aSIFSTime + aSlotTime
\]

\[
DIFS = 34 \mu s = aSIFSTime + 2 \times aSlotTime
\]

The EIFS is derived from the SIFS and the DIFS and the length of time it takes to transmit an ACK Control frame at PHY’s lowest mandatory rate by the following equation:

\[
EIFS \approx 92.7 \mu s = aSIFSTime + (8 \times ACKSize) + aPreambleLength + aPLCPHeaderLength + DIFS
\]

ACKSize is the length, in bytes, of an ACK frame; and \((8 \times ACKSize) + aPreambleLength + aPLCPHeaderLength\) is expressed in microseconds required to transmit at the PHY’s lowest mandatory rate, in 802.11a 6Mbit/s.
Appendix B, Abbreviations
A list of abbreviations to facilitate reading. Some abbreviations mean different things in the two standards. After abbreviations with double meaning is the standard written in brackets. (H2) means that the abbreviation is used in HiperLAN/2 and (802.11) that it is used in IEEEs standard.

ABIR ARQ Bandwidth Increase Request
ACF Association Control Function
ACH Access feedback CHannel
ACK acknowledgment
AID association identifier
AP Access Point
ARB ARQ feedback message request bit
ARQ Automatic Repeat reQuest
ASCH ASsociation control CHannel
ATIM announcement traffic indication message
BCCH Broadcast Control CHannel
BCH Broadcast Channel
BMB Bit Map Block
BMN Bit Map block Number
BPSK Binary Phase Shift Keying
BSA basic service area
BSS basic service set
BSSID basic service set identification
CAI Cumulative Acknowledgement Indicator
CC Central Controller
CCA clear channel assessment
CF contention free
CFP contention-free period
CID connection identifier
CL Convergence Layer
CM Centralized Mode
CP contention period
CRC Cyclic Redundancy Check (H2)
CRC cyclic redundancy code (802.11)
CS carrier sense
C-SAP Control Service Access Point
CTS clear to send
CW contention window
DA destination address
DBPSK differential binary phase shift keying
DCCH Dedicated Control CHannel
DCE data communication equipment
DCF distributed coordination function
DCLA direct current level adjustment
DES Data Encryption Standard
DFS Dynamic Frequency Selection
DIFS distributed (coordination function) interframe space
DiL Direct Link
DL DownLink
DLC Data Link Control
DLCC DLC Connection
DLL data link layer
DM Direct Mode
Dp desensitization
DQPSK differential quadrature phase shift keying
DS distribution system
DSAP destination service access point
DSM distribution system medium
DSS distribution system service
DSSS direct sequence spread spectrum
DTIM delivery traffic indication message
DUC DLC User Connection
DUCC DLC User Connection Control
EC Error Control
ED energy detection
EIFS extended interframe space
EIRP equivalent isotropically radiated power
ERS extended rate set
ESA extended service area
ESS extended service set
FC Flow Control (H2)
FC frame control (802.11)
FCCH Frame Control CHannel
FCH Frame CHannel
FCS frame check sequence
FER frame error ratio
FH frequency hopping
FHSS frequency-hopping spread spectrum
FIFO first in first out
GFSK Gaussian frequency shift keying
H/2 HIPERLAN type 2
IBSS independent basic service set
ICV integrity check value
IDU interface data unit
IE Information Element
IFS interframe space
IMp intermodulation protection
IR infrared
ISM industrial, scientific, and medical
IV Initialization Vector
LAN local area network
LCCH Link Control CHannel
LCH Long transport CHannel
LFSR Linear Feedback Shift Register
LLC logical link control
LME layer management entity
LRC long retry count
LSB Least Significant Bit
MAC ID MAC IDentifier
MAC Medium Access Control
MDF management-defined field
MIB management information base
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLME</td>
<td>MAC sublayer management entity</td>
</tr>
<tr>
<td>MMPDU</td>
<td>MAC management protocol data unit</td>
</tr>
<tr>
<td>MPDU</td>
<td>MAC protocol data unit</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>MSDU</td>
<td>MAC service data unit</td>
</tr>
<tr>
<td>MT</td>
<td>Mobile Terminal</td>
</tr>
<tr>
<td>N/A</td>
<td>not applicable</td>
</tr>
<tr>
<td>NAV</td>
<td>network allocation vector</td>
</tr>
<tr>
<td>NET ID</td>
<td>NETwork IDentifier</td>
</tr>
<tr>
<td>OFB</td>
<td>Output FeedBack mode</td>
</tr>
<tr>
<td>OFDM</td>
<td>Orthogonal Frequency Division Multiplexing</td>
</tr>
<tr>
<td>PC</td>
<td>point coordinator</td>
</tr>
<tr>
<td>PCF</td>
<td>point coordination function</td>
</tr>
<tr>
<td>PDU</td>
<td>protocol data unit</td>
</tr>
<tr>
<td>PDU Protocol Data Unit</td>
<td></td>
</tr>
<tr>
<td>PHY</td>
<td>physical (layer)</td>
</tr>
<tr>
<td>PHY-SAP</td>
<td>physical layer service access point</td>
</tr>
<tr>
<td>PIFS</td>
<td>point (coordination function) interframe space</td>
</tr>
<tr>
<td>PLCP</td>
<td>physical layer convergence protocol</td>
</tr>
<tr>
<td>PLME</td>
<td>physical layer management entity</td>
</tr>
<tr>
<td>PMD</td>
<td>physical medium dependent</td>
</tr>
<tr>
<td>PMD-SAP</td>
<td>physical medium dependent service access point</td>
</tr>
<tr>
<td>PN</td>
<td>pseudo-noise (code sequence)</td>
</tr>
<tr>
<td>PPDU</td>
<td>PLCP protocol data unit</td>
</tr>
<tr>
<td>ppm</td>
<td>parts per million</td>
</tr>
<tr>
<td>PPM</td>
<td>pulse position modulation</td>
</tr>
<tr>
<td>PRNG</td>
<td>pseudo-random number generator</td>
</tr>
<tr>
<td>PS</td>
<td>power save (mode)</td>
</tr>
<tr>
<td>PSDU</td>
<td>PLCP SDU</td>
</tr>
<tr>
<td>RA</td>
<td>receiver address</td>
</tr>
<tr>
<td>RACH</td>
<td>Random Access CHannel</td>
</tr>
<tr>
<td>RBCH</td>
<td>RLC Broadcast CHannel</td>
</tr>
<tr>
<td>RCH</td>
<td>Random CHannel</td>
</tr>
<tr>
<td>RF</td>
<td>radio frequency</td>
</tr>
</tbody>
</table>
UBCH User Broadcast Channel
UCT unconditional transition
UDCH User Data Channel
UL UpLink
UMCH User Multicast Channel
U-SAP User Service Access Point
WAN wide area network
WDM wireless distribution media
WDS wireless distribution system
WEP wired equivalent privacy
WM wireless medium
Appendix C, Source Code

In this appendix the source code for our Test program is included. The source code for the
driver that scans the PCI bus for a CardBus bridge and reads its registers is not included here.
The code is an important part of the programs function but is not necessary for understanding
of how it works. Anyone who is interested in the code is free to contact us.

**Short description of the files**

- **TEST** – starts the program
- **StdAfx** – file generated by Visual C++. Includes all files for the project
- **GivelO** – source code for the driver that gives the application direct port access
- **PCICIOC** – the link between the application and the pccd driver (pccd.sys which
not is included)
- **Socket** – methods for reading registers and scanning the bus for CardBus bridges
- **Portcom** – methods for reading from and writing to ports
- **TestDlg, BridgeDlg, ExCADlg, SocketDlg, IODlg and MemoryDlg** – files for the GUI

**TEST.h**

// TEST.h : main header file for the TEST application

#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h" // main symbols
#include "StdAfx.h"

// CTESTApp:
// See TEST.cpp for the implementation of this class

class CTESTApp : public CWinApp
{
public:
    CTESTApp();

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CTESTApp)
    public:
        virtual BOOL OnInitInstance();
    //}}AFX_VIRTUAL

// Implementation
    //{{AFX_MSG(CTESTApp)


// NOTE - the ClassWizard will add and remove member functions here.  
// DO NOT EDIT what you see in these blocks of generated code!
}AFX_MSG
DECLARE_MESSAGE_MAP()

/**************************************************************************/
//}}AFX_INSERT_LOCATION
// Microsoft Visual C++ will insert additional declarations immediately
before the previous line.

#endif // !defined(AFX_TEST_H__CFE4EED6_8F37_4658_8DF0_F9DCF2D89E55__INCLUDED_)

TEST.cpp
// TEST.cpp : Defines the class behaviors for the application.
/
#
#include "stdafx.h"
#include "TEST.h"
#include "TESTDlg.h"

#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/**************************************************************************/
// CTESTApp
BEGIN_MESSAGE_MAP(CTESTApp, CWinApp)
//}}AFX_MSG_MAP
// NOTE - the ClassWizard will add and remove mapping macros here.
// DO NOT EDIT what you see in these blocks of generated code!
}AFX_MSG
ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/**************************************************************************/
// CTESTApp construction

CTESTApp::CTESTApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

/**************************************************************************/
// The one and only CTESTApp object
CTESTApp theApp;
// CTESTApp initialization

BOOL CTESTApp::InitInstance()
{
    // Standard initialization
    // If you are not using these features and wish to reduce the size
    // of your final executable, you should remove from the following
    // the specific initialization routines you do not need.

    #ifdef _AFXDLL
        Enable3dControls(); // Call this when using MFC in a
                        // shared DLL
    #else
        Enable3dControlsStatic(); // Call this when linking to MFC
    #endif

    CTESTDlg dlg;
    m_pMainWnd = &dlg;
    int nResponse = dlg.DoModal();
    if (nResponse == IDOK)
    {
        // TODO: Place code here to handle when the dialog is
        // dismissed with OK
    }
    else if (nResponse == IDCANCEL)
    {
        // TODO: Place code here to handle when the dialog is
        // dismissed with Cancel
    }

    // Since the dialog has been closed, return FALSE so that we exit the
    // application, rather than start the application's message pump.
    return FALSE;
}

// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//
#if !defined(AFX_STDAFX_H__13373A7B_9D09_4CBF_94A1_366712AE822A__INCLUDED_)
#define AFX_STDAFX_H__13373A7B_9D09_4CBF_94A1_366712AE822A__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define VC_EXTRALEAN // Exclude rarely-used stuff from Windows
headers

#include <afxwin.h> // MFC core and standard components
#include <afxext.h> // MFC extensions
#include <afxctl.h> // MFC support for Internet Explorer 4 Common
Controls
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h> // MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT

#include "PCICIOC.h"
#include "socket.h"
#include "portcom.h"
#include <winsvc.h>
#include <winioctl.h>

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.
#endif // !defined(AFX_STDAFX_H__13373A7B_9D09_4CBF_94A1_366712AE822A__INCLUDED_)

StdAfx.cpp

// stdafx.cpp : source file that includes just the standard includes
// TEST.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information
#include "stdafx.h"

GiveIO.c

#include <ntddk.h>
#define DEVICE_NAME_STRING L"giveio"
#define IOPM_SIZE 0x2000
typedef UCHAR IOPM[IOPM_SIZE];

IOPM *IOPM_local = 0;
void Ke386SetIoAccessMap(int, IOPM *);
void Ke386QueryIoAccessMap(int, IOPM *);
void Ke386IoSetAccessProcess(PEPROCESS, int);

VOID GiveioUnload(IN PDRIVER_OBJECT DriverObject)
{
    WCHAR DOSNameBuffer[] = L"\DosDevices\" DEVICE_NAME_STRING;
    UNICODE_STRING uniDOSString;
    if(IOPM_local)
        MmFreeNonCachedMemory(IOPM_local, sizeof(IOPM));
    RtlInitUnicodeString(&uniDOSString, DOSNameBuffer);
    IoDeleteSymbolicLink (&uniDOSString);
    IoDeleteDevice(DriverObject->DeviceObject);
}

VOID SetIOPermissionMap(int OnFlag)
{
    Ke386IoSetAccessProcess(PsGetCurrentProcess(), OnFlag);
    Ke386SetIoAccessMap(1, IOPM_local);
}

void GiveIO(void)
{
    SetIOPermissionMap(1);
NTSTATUS GiveioCreateDispatch(IN PDEVICE_OBJECT DeviceObject,
   IN PIRP Irp)
{
    GiveIO(); // give the calling process I/O access

    Irp->IoStatus.Information = 0;
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

NTSTATUS DriverEntry(
   IN PDRIVER_OBJECT DriverObject,
   IN PUNICODE_STRING RegistryPath
)
{
    PDEVICE_OBJECT deviceObject;
    NTSTATUS status;
    WCHAR NameBuffer[] = L"\Device\" DEVICE_NAME_STRING;
    WCHAR DOSNameBuffer[] = L\DosDevices\DEVICE_NAME_STRING;
    UNICODE_STRING uniNameString, uniDOSString;

    // Allocate a buffer for the local IOPM and zero it.
    //
    IOPM_local = MmAllocateNonCachedMemory(sizeof(IOPM));
    if(IOPM_local == 0)
        return STATUS_INSUFFICIENT_RESOURCES;
    RtlZeroMemory(IOPM_local, sizeof(IOPM));

    // Set up device driver name and device object.
    //
    RtlInitUnicodeString(&uniNameString, NameBuffer);
    RtlInitUnicodeString(&uniDOSString, DOSNameBuffer);

    status = IoCreateDevice(DriverObject, 0,
                            &uniNameString, FILE_DEVICE_UNKNOWN, 0, FALSE,
                            &deviceObject);

    if(!NT_SUCCESS(status))
        return status;
    status = IoCreateSymbolicLink (&uniDOSString, &uniNameString);

    if (!NT_SUCCESS(status))
        return status;

    // Initialize the Driver Object with driver's entry points.
    // All we require are the Create and Unload operations.
    //
    DriverObject->MajorFunction[IRP_MJ_CREATE] = GiveioCreateDispatch;
    DriverObject->DriverUnload = GiveioUnload;
    return STATUS_SUCCESS;
}
# PCICIOC.h

// PCICIOC.H
// Common header file for DD and GUI. This files defines the
// IOCTL interface.

// Define the Device Type.
#define FILE_DEVICE_PCIC 0x83E0

// Define the I/O Commands the GUI sends to the DD.
#define IOCTL_PCIC_READ_REG (ULONG) CTL_CODE(FILE_DEVICE_PCIC, 0x01,
METHOD_IN_DIRECT, FILE_ANY_ACCESS)
#define IOCTL_PCIC_READ_REGS (ULONG) CTL_CODE(FILE_DEVICE_PCIC, 0x02,
METHOD_IN_DIRECT, FILE_ANY_ACCESS)
#define IOCTL_PCIC_WRITE_REG (ULONG) CTL_CODE(FILE_DEVICE_PCIC, 0x03,
METHOD_OUT_DIRECT, FILE_ANY_ACCESS)
#define IOCTL_PCIC_WRITE_REGS (ULONG) CTL_CODE(FILE_DEVICE_PCIC, 0x04,
METHOD_OUT_DIRECT, FILE_ANY_ACCESS)
#define IOCTL_PCI_READ_BYTE (ULONG) CTL_CODE(FILE_DEVICE_PCIC, 0x05,
METHOD_IN_DIRECT, FILE_ANY_ACCESS)
#define IOCTL_PCI_READ_WORD (ULONG) CTL_CODE(FILE_DEVICE_PCIC, 0x06,
METHOD_IN_DIRECT, FILE_ANY_ACCESS)
#define IOCTL_PCI_READ_DWORD (ULONG) CTL_CODE(FILE_DEVICE_PCIC, 0x07,
METHOD_IN_DIRECT, FILE_ANY_ACCESS)
#define IOCTL_PCI_READ_CONFIG (ULONG) CTL_CODE(FILE_DEVICE_PCIC, 0x08,
METHOD_IN_DIRECT, FILE_ANY_ACCESS)
#define IOCTL_PCI_WRITE_BYTE (ULONG) CTL_CODE(FILE_DEVICE_PCIC, 0x09,
METHOD_OUT_DIRECT, FILE_ANY_ACCESS)
#define IOCTL_PCI_WRITE_WORD (ULONG) CTL_CODE(FILE_DEVICE_PCIC, 0x0A,
METHOD_OUT_DIRECT, FILE_ANY_ACCESS)
#define IOCTL_PCI_WRITE_DWORD (ULONG) CTL_CODE(FILE_DEVICE_PCIC, 0x0B,
METHOD_OUT_DIRECT, FILE_ANY_ACCESS)
#define IOCTL_MAP_MEMORY (ULONG) CTL_CODE(FILE_DEVICE_PCIC, 0x10,
METHOD_IN_DIRECT, FILE_ANY_ACCESS)

// Typedef for mapping NT mapping physical address to virtual.
typedef struct
{
    ULONG physAddress;
    ULONG virtAddress;
} MAPMEM, *PMAPMEM;

// Typedef for CardBus info.
typedef struct
{
    USHORT pciBusDevFun;
    union
    {
        UCHAR ByteValue[256];
        USHORT WordValue[128];
        ULONG LongValue[64];
    } regs;
}
typedef struct
{
    USHORT BasePort;
    UCHAR OffsetReg;
    UCHAR Value;
    UCHAR *pExCA;
    UCHAR CardBus;
    PCARDBUS_BRIDGE_INFO pcbInfo;
} PCIC_REG, *PPCIC_REG;

typedef struct
{
    USHORT BasePort;
    UCHAR OffsetReg;
    UCHAR Values[0x40];
    UCHAR SocketRegValues[0x20];
    UCHAR Memory[0x3FF];
    UCHAR *pExCA;
    UCHAR *pCardMem;
    UCHAR CardBus;
    PCARDBUS_BRIDGE_INFO pcbInfo;
} PCIC_SOCKET, *PPCIC_SOCKET;

#define PCI_SIGNATURE 0x50434920 // "PCI 
#define IOCTL_PCIDD_BIOS 0x83B10001
#define IOCTL_PCIDD_MAPM 0x83B10002
#define PCI_FUNCTION_ID 0xB1
#define PCI_BIOS_PRESENT (PCI_FUNCTION_ID << 8) | 0x01
#define FIND_PCI_DEVICE (PCI_FUNCTION_ID << 8) | 0x02
#define FIND_PCI_CLASS_CODE (PCI_FUNCTION_ID << 8) | 0x03
#define GENERATE_SPECIAL_CYCLE (PCI_FUNCTION_ID << 8) | 0x06
#define READ_CONFIG_BYTE (PCI_FUNCTION_ID << 8) | 0x08
#define READ_CONFIG_WORD (PCI_FUNCTION_ID << 8) | 0x09
#define READ_CONFIG_DWORD (PCI_FUNCTION_ID << 8) | 0x0A
#define WRITE_CONFIG_BYTE (PCI_FUNCTION_ID << 8) | 0x0B
#define WRITE_CONFIG_WORD (PCI_FUNCTION_ID << 8) | 0x0C
#define WRITE_CONFIG_DWORD (PCI_FUNCTION_ID << 8) | 0x0D

typedef struct _REGS32
{
    ULONG eax;
    ULONG ebx;
    ULONG ecx;
    ULONG edx;
    ULONG esi;
    ULONG edi;
    ULONG Flags;
} REGS32, *PREGS32;
Socket.h

HANDLE InitRegIO();
void EndRegIO(HANDLE hPcie);
BOOL ReadSocket(HANDLE hPcie, PPCIC_SOCKET pPcieSocket);
BOOL WriteSocket(HANDLE hPcie, PPCIC_SOCKET pPcieSocket);
BOOL ReadPciByte(HANDLE hPcie, PCARDBUS_BRIDGE_INFO pcbInfo, int iOffset);
BOOL ReadPciWord(HANDLE hPcie, PCARDBUS_BRIDGE_INFO pcbInfo, int iOffset);
BOOL ReadPciDWord(HANDLE hPcie, PCARDBUS_BRIDGE_INFO pcbInfo, int iOffset);
BOOL MapExCA(HANDLE hPcie, PPCIC_SOCKET pPcieSocket);
BOOL MapCardMem(HANDLE hPcie, PPCIC_SOCKET pPcieSocket);

#define MAX_SOCKETS 4
#define SOCKET_A 100
#define SOCKET_B 101
#define SOCKET_C 102
#define SOCKET_D 103

BOOL InitBridge(HANDLE hPcie, PCIC_SOCKET PcicSocket[]);

Socket.cpp

#include "stdafx.h"
#include "resource.h"

BOOL InstallDriver(SC_HANDLE schSCManager);
BOOL StartDriver(SC_HANDLE schSCManager);
BOOL StopDriver(SC_HANDLE schSCManager);
BOOL RemoveDriver(SC_HANDLE schSCManager);

OSVERSIONINFO osVer;

int NumSockets = 0; // default to 0 sockets.
CARDBUS_BRIDGE_INFO cbInfo[MAX_SOCKETS];

int CurrentSocket = 0; // 0=A, 1=B, etc.
int CurrentWindow = 0; // 0=Gen, 1=IRQ/IO, 2=Mem

BOOL flLegacy = FALSE;
BOOL flCardBus = FALSE;

BOOL InitBridge(HANDLE hPcie, PCIC_SOCKET PcicSocket[]){

    int iSocket, iBus, iDev, iFun;
    int iOffset;

    // Now look for CardBus controllers. Loop over 2 buses, 32 devices,
    // 8 functions.
    iSocket = 0;
    for (iBus=0; iBus<2; iBus++)
for (iDev=0; iDev<32; iDev++)
{
for (iFun=0; iFun<8; iFun++)
{
    cbInfo[iSocket].pciBusDevFun = (iBus << 8) | (iDev << 3) | iFun;
    ReadPciDWord(hPcic, &cbInfo[iSocket], 8);
    //check if cardbus-to-pci bridge
    if ((cbInfo[iSocket].regs.LongValue[2] & 0xFFFFFF00) == 0x06070000)
    {
        flCardBus = TRUE;
        PcicSocket[iSocket].CardBus = FALSE;
        PcicSocket[iSocket].pcbInfo = &cbInfo[iSocket];
        // Read CardBus configuration registers.
        for (iOffset=0; iOffset<256; iOffset+=4)
            ReadPciDWord(hPcic, &cbInfo[iSocket], iOffset);
        // If we're not in Legacy Mode, map the ExCA register
        // memory pointer and read the socket.
        if (!flLegacy)
        {
            MapExCA(hPcic, &PcicSocket[iSocket]);
            PcicSocket[iSocket].CardBus = TRUE;
            //copy ExCA from pci-mem to Values
            memcpy(&PcicSocket[iSocket].Values,
                   PcicSocket[iSocket].pExCA+0x800, 0x40);
            //copy SocketRegs from pci-mem to SocketRegsValues
            memcpy(&PcicSocket[iSocket].SocketRegValues,
                   PcicSocket[iSocket].pExCA, 0x20);
            PcicSocket[iSocket].OffsetReg = iSocket*0x40;
            //Map the cards memory space
            MapCardMem(hPcic, &PcicSocket[iSocket]);
        }
        iSocket++;
    }
}
return 1;

HANDLE InitRegIO()
{
    HANDLE hPcic;
    SC_HANDLE schSCManager;
    osVer.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
    GetVersionEx(&osVer);
    if (osVer.dwPlatformId == VER_PLATFORM_WIN32_NT)
    {
        //
// Install the NT device driver PcitDD.sys
schSCManager = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
hPcit = INVALID_HANDLE_VALUE;
if (InstallDriver(schSCManager))
    if (StartDriver(schSCManager))
       ’hPcit = CreateFile("\\\.\PcitDD",
            GENERIC_READ, FILE_SHARE_READ,
            NULL, OPEN_EXISTING, 0, NULL);
    CloseServiceHandle(schSCManager);
} return hPcit;

void EndRegIO(HANDLE hPcit)
{
SC_HANDLE schSCManager;
CloseHandle(hPcit);
if (osVer.dwPlatformId == VER_PLATFORM_WIN32_NT)
    { schSCManager = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
        StopDriver(schSCManager);
        RemoveDriver(schSCManager);
        CloseServiceHandle(schSCManager);
    }

BOOL ReadSocket(HANDLE hPcit, PPCIC_SOCKET pPcitSocket)
{
    DWORD dwRet;

    if (osVer.dwPlatformId == VER_PLATFORM_WIN32_NT)
        { if (pPcitSocket->CardBus)
            { int i;
                for (i=0; i<0x40; i++)
                    pPcitSocket->Values[i] = *((pPcitSocket->pExCA)+0x800+i);
            } else
                DeviceIoControl(hPcit, IOCTL_PCIC_READ_REGS, pPcitSocket,
                    sizeof(PCIC_SOCKET), pPcitSocket,
                    sizeof(PCIC_SOCKET), &dwRet, NULL);
        } return TRUE;

BOOL WriteSocket(HANDLE hPcit, PPCIC_SOCKET pPcitSocket)
{
    DWORD dwRet;

    if (osVer.dwPlatformId == VER_PLATFORM_WIN32_NT)
        { if (pPcitSocket->CardBus)
            {
int i;

for (i=0; i<0x40; i++)
    *((pPcicSocket->pExCA)+0x800+i) = pPcicSocket->Values[i];
} else
    DeviceIoControl(hPcic, IOCTL_PCIC_WRITE_REGS, pPcicSocket,
                    sizeof(PCIC_SOCKET), pPcicSocket,
                    sizeof(PCIC_SOCKET),
                    &dwRet, NULL);
}
return TRUE;

BOOL InstallDriver(SC_HANDLE schSCManager)
{
    SC_HANDLE schService;
    DWORD dwErr;
    char driverSys[128];
    WIN32_FIND_DATA w32FindData;
    HANDLE hFileFind;

    // Look for PCICDD.SYS. Start at local directory. If not found,
    // look in Windows system drivers directory. If not found, its an
    // error!
    GetCurrentDirectory(128, driverSys);
    strcat(driverSys, "\pcicdd.sys");
    hFileFind = FindFirstFile(driverSys, &w32FindData);
    if (hFileFind == INVALID_HANDLE_VALUE)
    {
        GetSystemDirectory(driverSys, 128);
        strcat(driverSys, "\drivers\pcicdd.sys");
        hFileFind = FindFirstFile(driverSys, &w32FindData);
        if (hFileFind == INVALID_HANDLE_VALUE)
        {
            MessageBox(HWND_DESKTOP, "Cannot find PCICDD.SYS.",
                        "WinPCIC - Error", MB_OK | MB_ICONERROR);
            return FALSE;
        }
    }

    schService = CreateService(schSCManager,
                                "pcicdd",
                                "pcicdd",
                                SERVICE_ALL_ACCESS,
                                SERVICE_KERNEL_DRIVER,
                                SERVICE_DEMAND_START,
                                SERVICE_ERROR_NORMAL,
                                driverSys,
                                NULL,
                                NULL,
                                NULL,
                                NULL);

    if (schService == NULL)
    {
        dwErr = GetLastErrorCode();
        if (dwErr == ERROR_SERVICE_EXISTS)
            return TRUE;
        else
...
BOOL StartDriver(SC_HANDLE schSCManager)
{
    SC_HANDLE schService;
    BOOL flRet;
    DWORD dwErr;

    schService = OpenService(schSCManager, "pcicdd", SERVICE_ALL_ACCESS);
    if (schService == NULL)
    {
        MessageBox(HWND_DESKTOP, "Cannot open PCICDD.SYS.",
                   "WinPCIC - Error", MB_OK | MB_ICONERROR);
        return FALSE;
    }

    flRet = StartService(schService, 0, NULL);
    if (!flRet)
    {
        dwErr = GetLastError();
        if (dwErr != ERROR_SERVICE_ALREADY_RUNNING)
            MessageBox(HWND_DESKTOP, "Cannot start PCICDD.SYS.",
                        "WinPCIC - Error", MB_OK | MB_ICONERROR);
        else
            flRet = TRUE;
    }

    CloseServiceHandle(schService);
    return flRet;
}

BOOL StopDriver(SC_HANDLE schSCManager)
{
    SC_HANDLE schService;
    BOOL flRet;
    SERVICE_STATUS serviceStatus;

    schService = OpenService(schSCManager, "pcicdd", SERVICE_ALL_ACCESS);
    if (schService != NULL)
    {
        flRet = ControlService(schService, SERVICE_CONTROL_STOP,
                               &serviceStatus);
        if (flRet)
            CloseServiceHandle(schService);
        else
            flRet = FALSE;
    }

    return flRet;
}
BOOL RemoveDriver(SC_HANDLE schSCManager)
{
    SC_HANDLE schService;
    BOOL flRet;

    schService = OpenService(schSCManager, "pcicdd", SERVICE_ALL_ACCESS);
    if (schService != NULL)
    {
        flRet = DeleteService(schService);
        CloseServiceHandle(schService);
    }
    else
    {
        flRet = FALSE;
    }

    return flRet;
}

BOOL ReadPciByte(HANDLE hPcic, PCARDBUS_BRIDGE_INFO pcbInfo, int iOffset)
{
    REGS32 regs32;
    DWORD dwRet;

    regs32.eax = READ_CONFIG_BYTE;
    regs32.edi = iOffset;
    regs32.ebx = pcbInfo->pciBusDevFun;
    regs32.ecx = 0;
    if (osVer.dwPlatformId == VER_PLATFORM_WIN32_NT)
    {
        DeviceIoControl(hPcic, IOCTL_PCI_READ_BYTE, &regs32, sizeof(REGS32),
                        &regs32, sizeof(REGS32), &dwRet, NULL);
        pcbInfo->regs.ByteValue[iOffset] = (UCHAR) regs32.ecx;
    }
    return TRUE;
}

BOOL ReadPciWord(HANDLE hPcic, PCARDBUS_BRIDGE_INFO pcbInfo, int iOffset)
{
    REGS32 regs32;
    DWORD dwRet;

    regs32.eax = READ_CONFIG_WORD;
    regs32.edi = iOffset;
    regs32.ebx = pcbInfo->pciBusDevFun;
    regs32.ecx = 0;
    if (osVer.dwPlatformId == VER_PLATFORM_WIN32_NT)
    {
        DeviceIoControl(hPcic, IOCTL_PCI_READ_WORD, &regs32, sizeof(REGS32),
                        &regs32, sizeof(REGS32), &dwRet, NULL);
        pcbInfo->regs.WordValue[iOffset/2] = (USHORT) regs32.ecx;
    }
    return TRUE;
}

BOOL ReadPciDWord(HANDLE hPcic, PCARDBUS_BRIDGE_INFO pcbInfo, int iOffset)
{
    REGS32 regs32;
    DWORD dwRet;

    regs32.eax = 0;
    regs32.edi = iOffset;
    regs32.ebx = pcbInfo->pciBusDevFun;
    regs32.ecx = 0;
    if (osVer.dwPlatformId == VER_PLATFORM_WIN32_NT)
    {
        DeviceIoControl(hPcic, IOCTL_PCI_READ_DWORD, &regs32, sizeof(REGS32),
                        &regs32, sizeof(REGS32), &dwRet, NULL);
        pcbInfo->regs.DWordValue[iOffset/4] = (ULONG) regs32.ecx;
    }
    return TRUE;
}
regs32 eax = READ_CONFIG_DWORD;
regs32 edi = iOffset;
regs32 ebx = pcbInfo->pciBusDevFun;
regs32 ecx = 0;
if (osVer.dwPlatformId == VER_PLATFORM_WIN32_NT) {
    DeviceIoControl(hPcic, IOCTL_PCI_READ_DWORD, &regs32, sizeof(REGS32),
                    &regs32, sizeof(REGS32), &dwRet, NULL);
    pcbInfo->regs.LongValue[iOffset/4] = (ULONG) regs32.ecx;
}
return TRUE;

BOOL WritePciByte(HANDLE hPcic, PCARDBUS_BRIDGE_INFO pcbInfo, int
iOffset) {
    REGS32 regs32;
    DWORD dwRet;
    regs32 eax = WRITE_CONFIG_BYTE;
    regs32 edi = iOffset;
    regs32 ebx = pcbInfo->pciBusDevFun;
    regs32 ecx = pcbInfo->regs.ByteValue[iOffset];
    if (osVer.dwPlatformId == VER_PLATFORM_WIN32_NT) {
        DeviceIoControl(hPcic, IOCTL_PCI_WRITE_BYTE, &regs32, sizeof(REGS32),
                        &regs32, sizeof(REGS32), &dwRet, NULL);
    }
    return TRUE;
}

BOOL WritePciWord(HANDLE hPcic, PCARDBUS_BRIDGE_INFO pcbInfo, int
iOffset) {
    REGS32 regs32;
    DWORD dwRet;
    regs32 eax = WRITE_CONFIG_WORD;
    regs32 edi = iOffset;
    regs32 ebx = pcbInfo->pciBusDevFun;
    regs32 ecx = pcbInfo->regs.WordValue[iOffset/2];
    if (osVer.dwPlatformId == VER_PLATFORM_WIN32_NT) {
        DeviceIoControl(hPcic, IOCTL_PCI_WRITE_WORD, &regs32, sizeof(REGS32),
                        &regs32, sizeof(REGS32), &dwRet, NULL);
    }
    return TRUE;
}

BOOL WritePciDWord(HANDLE hPcic, PCARDBUS_BRIDGE_INFO pcbInfo, int
iOffset) {
    REGS32 regs32;
    DWORD dwRet;
    regs32 eax = WRITE_CONFIG_DWORD;
    regs32 edi = iOffset;
    regs32 ebx = pcbInfo->pciBusDevFun;
    regs32 ecx = pcbInfo->regs.LongValue[iOffset/4];
    if (osVer.dwPlatformId == VER_PLATFORM_WIN32_NT)
BOOL MapExCA(HANDLE hPcic, PPCIC_SOCKET pPcicSocket)
{
    DWORD dwRet;
    MAPMEM mapMem; // for NT

    if (osVer.dwPlatformId == VER_PLATFORM_WIN32_NT)
    {
        // Get the ExCA Base Address and map it to a linear address.
        ReadPciDWord(hPcic, pPcicSocket->pcbInfo, 0x10);
        mapMem.physAddress = pPcicSocket->pcbInfo->regs.LongValue[4];
        DeviceIoControl(hPcic, IOCTL_MAP_MEMORY, &mapMem, 4,
                        &mapMem, 4, &dwRet, NULL);
        pPcicSocket->pExCA = (UCHAR *) mapMem.virtAddress;
    }
    return TRUE;
}

BOOL MapCardMem(HANDLE hPcic, PPCIC_SOCKET pPcicSocket)
{
    DWORD dwRet;
    MAPMEM mapMem; // for NT

    if (osVer.dwPlatformId == VER_PLATFORM_WIN32_NT)
    {
        // Get the Memory Base Address and map it to a linear address.
        //ReadPciDWord(hPcic, pPcicSocket->pcbInfo, 0x10);
        mapMem.physAddress = 0x000FFFFF & ((pPcicSocket->Values[0x10])<< 12);
        DeviceIoControl(hPcic, IOCTL_MAP_MEMORY, &mapMem, 4,
                        &mapMem, 4, &dwRet, NULL);
        pPcicSocket->pCardMem = (UCHAR *) mapMem.virtAddress;
    }
    return TRUE;
}

Portcom.h

// Writes a UINT value to a port (Not used in this version)
void outport(UINT portid, UINT value);
// Writes a BYTE value to a port (Not used in this version)
void outportb(UINT portid, BYTE value);

// Reads a Port and returns a BYTE value
BYTE inportb(UINT portid);
// Reads a Port and returns a UINT value
UINT inport(UINT portid);

HANDLE InitPortIO(); // returns handle to giveio-device
Portcom.cpp

#include "stdafx.h"
#include "resource.h"

BOOL InstallIODriver(SC_HANDLE schSCManager);
BOOL StartIODriver(SC_HANDLE schSCManager);
BOOL StopIODriver(SC_HANDLE schSCManager);
BOOL RemoveIODriver(SC_HANDLE schSCManager);

void outport(UINT portid, UINT value)
{
__asm mov edx,portid;
__asm mov eax,value;
__asm out dx,ax;
}

void outportb(UINT portid, BYTE value)
{
__asm mov edx,portid
__asm mov al,value
__asm out dx,al
}

BYTE inportb(UINT portid)
{
unsigned char value;
__asm mov edx,portid
__asm in al,dx
__asm mov value,al
return value;
}

UINT inport(UINT portid)
{
int value=0;
__asm mov edx,portid
__asm in ax,dx
__asm mov value,eax
return value;
}

HANDLE InitPortIO()
{
HANDLE h;
SC_HANDLE schSCManager;

//osVer.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
//GetVersionEx(&osVer);

//if (osVer.dwPlatformId == VER_PLATFORM_WIN32_NT)
//{
  //open device
  // Install the NT device driver GIVEIO.sys
  schSCManager = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
  h = INVALID_HANDLE_VALUE;
  if (InstallIODriver(schSCManager))
if (StartIODriver(schSCManager))
    h = CreateFile("\\\.\giveio",
        GENERIC_READ, FILE_SHARE_READ,
        NULL, OPEN_EXISTING, 0, NULL);

CloseServiceHandle(schSCManager);
//}}
return h;
}

BOOL InstallIODriver(SC_HANDLE schSCManager)
{
    SC_HANDLE schService;
    DWORD dwErr;
    char driverSys[128];
    WIN32_FIND_DATA w32FindData;
    HANDLE hFileFind;

    // Look for GIVEIO.SYS. Start at local directory. If not found,
    // look in Windows system drivers directory. If not found, its
    // an error!
    GetCurrentDirectory(128, driverSys);
    strcat(driverSys, "\\giveio.sys");
    hFileFind = FindFirstFile(driverSys, &w32FindData);
    if (hFileFind == INVALID_HANDLE_VALUE)
    {
        GetSystemDirectory(driverSys, 128);
        strcat(driverSys, "\\drivers\giveio.sys");
        hFileFind = FindFirstFile(driverSys, &w32FindData);
        if (hFileFind == INVALID_HANDLE_VALUE)
        {
            MessageBox(HWND_DESKTOP, "Cannot find GIVEIO.SYS.",
                "WinPCIC - Error", MB_OK | MB_ICONERROR);
            return FALSE;
        }
    }
    schService = CreateService(schSCManager,
        "giveio",
        "giveio",
        SERVICE_ALL_ACCESS,
        SERVICE_KERNEL_DRIVER,
        SERVICE_DEMAND_START,
        SERVICE_ERROR_NORMAL,
        driverSys,
        NULL,
        NULL,
        NULL,
        NULL);

    if (schService == NULL)
    {
        dwErr = GetLastError();
        if (dwErr == ERROR_SERVICE_EXISTS)
            return TRUE;
        else
        {
            MessageBox(HWND_DESKTOP, "Cannot install GIVEIO.SYS.",
                "WinPCIC - Error", MB_OK | MB_ICONERROR);
            return FALSE;
        }
    }
}

BOOL StopIODriver(SC_HANDLE schSCManager)
{
    SC_HANDLE schService;
    BOOL flRet;
    SERVICE_STATUS serviceStatus;

    schService = OpenService(schSCManager, "giveio", SERVICE_ALL_ACCESS);
    if (schService != NULL)
    {
        flRet = ControlService(schService, SERVICE_CONTROL_STOP, &serviceStatus);
        CloseServiceHandle(schService);
    }
    else
    {
        flRet = FALSE;
    }
    return flRet;
}

BOOL RemoveIODriver(SC_HANDLE schSCManager)
{
    SC_HANDLE schService;
    BOOL flRet;

    return TRUE;
}
schService = OpenService(schSCManager, "giveio", SERVICE_ALL_ACCESS);
if (schService != NULL)
{
    flRet = DeleteService(schService);
    CloseServiceHandle(schService);
} else
    flRet = FALSE;
return flRet;

BridgeDlg.h
#if !defined(AFX_BRIDGEDLG_H__0757211D_3B1F_4627_BE40_F0D6976BACDC__INCLUDED_)
define AFX_BRIDGEDLG_H__0757211D_3B1F_4627_BE40_F0D6976BACDC__INCLUDED_
#endif // _MSC_VER > 1000
#pragma once
// BridgeDlg.h : header file

///////////////////////////////////////////////////////////////////////////
// BridgeDlg dialog

class BridgeDlg : public CDialog
{
// Construction
public:
    BridgeDlg(CWnd* pParent = NULL); // standard constructor

// Dialog Data
    enum { IDD = IDD_BRIDGE };

// Overrides
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV

// Implementation
protected:
    DECLARE_MESSAGE_MAP()

};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately
before the previous line.
BridgeDlg.cpp

// BridgeDlg.cpp : implementation file

#include "stdafx.h"
#include "TEST.h"
#include "BridgeDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

///////////////////////////////////////////////////////////////////////////
//
// BridgeDlg dialog

BridgeDlg::BridgeDlg(CWnd* pParent /*=NULL*/)
: CDialog(BridgeDlg::IDD, pParent)
{
    //AFX_DATA_INIT(BridgeDlg)
    }

void BridgeDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //AFX_DATA_MAP(BridgeDlg)
    }

BEGIN_MESSAGE_MAP(BridgeDlg, CDialog)
    //AFX_MSG_MAP(BridgeDlg)
    
END_MESSAGE_MAP()

///////////////////////////////////////////////////////////////////////////

EXCADlg.h

#if
!defined(AFX_EXCADLG_H__A839C5F0_16B2_447E_9CAD_79D1B2B69A71__INCLUDED_)
define AFX_EXCADLG_H__A839C5F0_16B2_447E_9CAD_79D1B2B69A71__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

EXCADlg.h : header file

87
// EXCADlg dialog

class EXCADlg : public CDialog
{
  // Construction
  public:
      EXCADlg(CWnd* pParent = NULL); // standard constructor

  // Dialog Data
  //{{AFX_DATA(EXCADlg)
  enum { IDD = IDD_EXCA };
 //}}AFX_DATA

  // Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(EXCADlg)
  protected:
      virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
  //}}AFX_VIRTUAL

  // Implementation
  protected:
      BOOL OnInitDialog();
  // Generated message map functions
  //{{AFX_MSG(EXCADlg)
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

//EXCADlg.cpp
// EXCADlg.cpp : implementation file
//
#include "stdafx.h"
#include "TEST.h"
#include "EXCADlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

static char THIS_FILE[] = __FILE__;

Excadlg.cpp
// EXCADlg dialog

EXCADlg::EXCADlg(CWnd* pParent /*=NULL*/)
: CDlg(EXCADlg::IDD, pParent)
{
    //AFX_DATA_INIT(EXCADlg)
    //}AFX_DATA_INIT
}

BOOL EXCADlg::OnInitDialog()
{
    CDlg::OnInitDialog();
    UpdateData(false);
    return false;
}

void EXCADlg::DoDataExchange(CDataExchange* pDX)
{
    CDlg::DoDataExchange(pDX);
    //AFX_DATA_MAP(EXCADlg)
    //}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(EXCADlg, CDlg)
    //AFX_MSG_MAP(EXCADlg)
    //}AFX_MSG_MAP
END_MESSAGE_MAP()

///////////////////////////////////////////////////////////////////////////
// EXCADlg message handlers

IODlg.h
#if !defined(AFX_IODLG_H__98EC9C3C_B5D1_4CB1_8641_B1AC8E999852__INCLUDED_)
#define AFX_IODLG_H__98EC9C3C_B5D1_4CB1_8641_B1AC8E999852__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

// IODlg.h : header file

.parallel

BEGIN_MESSAGE_MAP(IODlg, CDialog)
    //AFX_MSG_MAP(IODlg)
    //}AFX_MSG_MAP
END_MESSAGE_MAP()

///////////////////////////////////////////////////////////////////////////
// IODlg message handlers

class IODlg : public CDialog
{
    // Construction
public:
    IODlg(CWnd* pParent = NULL); // standard constructor

    void PublicIO (int startport, int endpoint);
    // Dialog Data
    //AFX_DATA(IODlg)
    enum { IDD = IDD_IO };
};
// Overrides
// ClassWizard generated virtual function overrides
//}}AFX_VIRTUAL(IODlg)
protected:
virtual void DoDataExchange(CDataExchange* pDX);
 // DDX/DDV support
//}}AFX_VIRTUAL

// Implementation
protected:
void IO (int startport, int endport);

// Generated message map functions
//}}AFX_MSG(IODlg)
DECLARE_MESSAGE_MAP()

}{AFX_INSERT_LOCATION}
// Microsoft Visual C++ will insert additional declarations immediately
before the previous line.

#endif // !defined(AFX_IODLG_H__98EC9C3C_B5D1_4CB1_8641_B1AC8E999852__INCLUDED_)

IODlg.cpp
// IODlg.cpp : implementation file
/

#include "stdafx.h"
#include "TEST.h"
#include "IODlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

polator:// IODlg dialog

IODlg::IODlg(CWnd* pParent /*=NULL*/)
: CDialog(IODlg::IDD, pParent)
{
//}}AFX_DATA_INIT

void IODlg::DoDataExchange(CDataExchange* pDX)
{

}
CDlg::DoDataExchange(pDX);
//{{AFX_DATA_MAP(IODlg)
DDX_Control(pDX, IDC_IO_LIST2, m_ctlIOList2);
DDX_Control(pDX, IDC_IO_LIST, m_ctlIOList);
//}}AFX_DATA_MAP

BEGIN_MESSAGE_MAP(IODlg, CDialog)
//{{AFX_MSG_MAP(IODlg)
// NOTE: the ClassWizard will add message map macros here
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

///////////////////////////////////////////////////////////////////////////
//
// IODlg message handlers

void IODlg::IO (int startport, int endport){
    m_ctlIOList.ResetContent();
    m_ctlIOList2.ResetContent();
    char Hex [2];
    int i;
    UCHAR value;
    CString AllBytes;
    for (i=startport;i<=endport;i++){
        value = inportb(i);
        wsprintf(Hex, "%2.2X", value);
        AllBytes.Format("%d", i);
        AllBytes +=": ";
        AllBytes += Hex;
        if (i<(endport-startport)/2+startport)
            m_ctlIOList.AddString(AllBytes);
        else
            m_ctlIOList2.AddString(AllBytes);
        AllBytes.Empty();
    }
}

void IODlg::PublicIO (int startport, int endport){
    IO(startport, endport);
}

MemoryDlg.h
#if !defined(AFX_MEMORYDLG_H__340081BA_80F0_47D9_8854_1E566E5ADBF0__INCLUDED_
#define AFX_MEMORYDLG_H__340081BA_80F0_47D9_8854_1E566E5ADBF0__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// MemoryDlg.h : header file
/

////////////////////////////////////////////////////////////////////////////////
//
// MemoryDlg dialog
class MemoryDlg : public CDialog
{
    // Construction
    public:
        MemoryDlg(CWnd* pParent = NULL); // standard constructor
    void PublicOnHex(PPCIC_SOCKET PcicSocket);
    void PublicOnChar(PPCIC_SOCKET PcicSocket);
    PPCIC_SOCKET PcicSocket2;
    // Dialog Data
    {{AFX_DATA(MemoryDlg)
        enum { IDD = IDD_MEMORY };
        CListBox m_ctlMemList;
        int m_intRadio;
    }}AFX_DATA
    // Overrides
    // ClassWizard generated virtual function overrides
    {{AFX_VIRTUAL(MemoryDlg)
        protected:
            virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
    }}AFX_VIRTUAL
    // Implementation
    protected:
        // Generated message map functions
        {{AFX_MSG(MemoryDlg)
            afx_msg void OnChar();
            afx_msg void OnHex();
        }}AFX_MSG
        DECLARE_MESSAGE_MAP()
    
};

('{{AFX_INSERT_LOCATION}})

#ifdef !_DEBUG
#define new DEBUG_NEW
#endif
static char THIS_FILE[] = __FILE__;

MemoryDlg.cpp
// MemoryDlg.cpp : implementation file

#include "stdafx.h"
#include "TEST.h"
#include "MemoryDlg.h"

#include "stdafx.h"
#include "TEST.h"
#include "MemoryDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

static char THIS_FILE[] = __FILE__;

92
MemoryDlg::MemoryDlg(CWnd* pParent /*=NULL*/)
: CDialog(MemoryDlg::IDD, pParent)
{
    //AFX_DATA_INIT(MemoryDlg)
    m_intRadio = 0;
    //}AFX_DATA_INIT

}

void MemoryDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //AFX_DATA_MAP(MemoryDlg)
    DDX_Control(pDX, IDC_MEM_LIST, m_ctlMemList);
    DDX_Radio(pDX, IDC_RADIO1, m_intRadio);
    //}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(MemoryDlg, CDialog)
    //AFX_MSG_MAP(MemoryDlg)
    ON_BN_CLICKED(IDC_RADIO2, OnChar)
    ON_BN_CLICKED(IDC_RADIO1, OnHex)
    //}AFX_MSG_MAP
END_MESSAGE_MAP()

_OVERRIDE_ void MemoryDlg::OnChar()
{
    int Row, Tuple;
    char Char[2];
    CString AllBytes;
    m_ctlMemList.ResetContent();
    for (Row = 0; Row < 64; Row++){
        AllBytes.Format("%x", Row*16);
        AllBytes.Insert(0, "0x");
        AllBytes += ": ";
        for (Tuple = 0; Tuple < 16; Tuple++){
            wsprintf(Char, "%C",
                    *(PcieSocket2[0].pCardMem+Tuple+Row*16));
            AllBytes += Char;
        }
        //AllBytes += Row + ": ";
        m_ctlMemList.AddString(AllBytes);
        AllBytes.Empty();
    }
}

_OVERRIDE_ void MemoryDlg::OnHex()
{
    int Row, Tuple;
    char Hex[2];
    CString AllBytes;
    m_ctlMemList.ResetContent();
    for (Row = 0; Row < 64; Row++){
        AllBytes.Format("%x", Row*16);
        //
    }

}
AllBytes.Insert (0, "0x");
AllBytes += " ";
for (Tuple=0; Tuple<16; Tuple++){
    wsprintf(Hex, "%2.2X",
        *(PcicSocket2[0].pCardMem+Tuple+Row*16));
    AllBytes += Hex;
    AllBytes += " ";
}
m_ctlMemList.AddString(AllBytes);
AllBytes.Empty();
}
void MemoryDlg::PublicOnHex(PPCIC_SOCKET PcicSocket){
PcicSocket2 = PcicSocket;
OnHex();
}
void MemoryDlg::PublicOnChar(PPCIC_SOCKET PcicSocket){
PcicSocket2 = PcicSocket;
OnChar();
}

SocketDlg.h

#if !defined(AFX_SOCKETDLG_H__CA180D98_8210_4383_8BE8_F47841620B68__INCLUDED_)
#define AFX_SOCKETDLG_H__CA180D98_8210_4383_8BE8_F47841620B68__INCLUDED_
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// SocketDlg.h : header file
///< SocketDlg dialog
class SocketDlg : public CDialog
{
    // Construction
    public:
        SocketDlg(CWnd* pParent = NULL); // standard constructor
    // Dialog Data
        // ClassWizard generated virtual function overrides
       //}}AFX_DATA
        // Overides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(SocketDlg)
        protected:
            virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV
        support
       //}}AFX_VIRTUAL
SocketDlg.cpp

// SocketDlg.cpp : implementation file

#include "stdafx.h"
#include "TEST.h"
#include "SocketDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

///////////////////////////////////////////////////////////////////////////
// SocketDlg dialog

SocketDlg::SocketDlg(CWnd* pParent /*=NULL*/)
    : CDialog(SocketDlg::IDD, pParent)
{
    //AFX_DATA_INIT(SocketDlg)
    }

void SocketDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //AFX_DATA_MAP(SocketDlg)
    }

BEGIN_MESSAGE_MAP(SocketDlg, CDialog)
    //AFX_MSG_MAP(SocketDlg)
    // NOTE: the ClassWizard will add DDX and DDV calls here
    //AFX_MSG_MAP
END_MESSAGE_MAP()
Appendix D, Program flowchart

Start
Main-program

Enable direct
port access

Success?
No  Show Dialog box
with error  End program

Load driver for
reading PCI bus

Success?
No  Show Dialog box
with error  End program

Read registers
and store values

Show GUI

Button
pressed?
No  Exit button  End program

Tab button  Change GUI
Dialog