Kapitel 1
Die MAC-(Medium Access Control) Teilschicht

1.1 Architektur eines bus-basierten Multicomputersystems

In einem busbasierten Multicomputersystem lassen sich eine Anzahl von Rechnern so miteinander verbinden, daß beliebige Datenmengen in wenigen Millisekunden übertragen werden können. Die Topologie eines solchen Systems ist relativ einfach und im folgenden Bild illustriert:

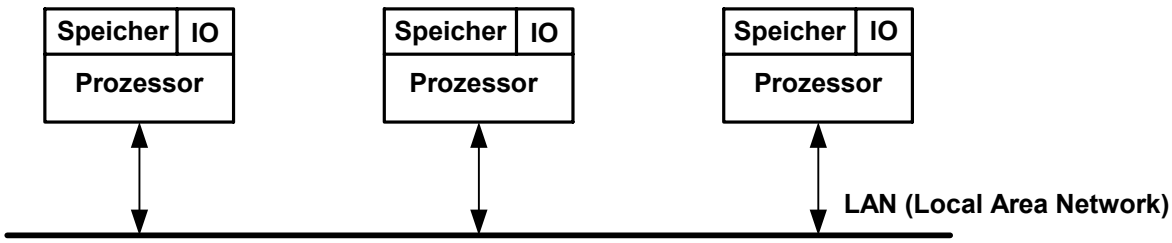


Bild 1: Topologie eines Multicomputersystems

Jeder Prozessor hat seinen eigenen lokalen Speicher und besitzt sein eigenes lokales IO-System (analoge und digitale Ein-/Ausgabe, Interrupt-Controller, Timer usw.). Die Prozessoren sind über ein lokales Netzwerk miteinander verbunden, und die spannende Frage ist: Wie können die Prozessoren Informationen austauschen? Sie benutzen UARTs (Universal Asynchron Receive Transmit), die zwischen ihnen und dem LAN liegen. Hierzu ein Beispiel, siehe Bild 2:

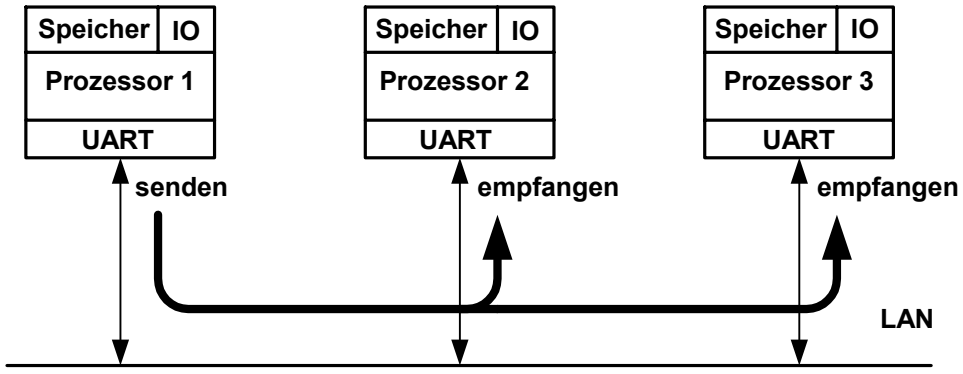


Bild 2: Kommunikation über UARTs

Prozessor 1 schickt über seinen UART eine Botschaft ins Netzwerk, und die Prozessoren 2 und 3 empfangen sie von dort über ihre UARTs.
Es sind also die UARTs, die im Physical Layer (Layer 1) den Informationsaustausch zwischen den Rechnern realisieren.

Wie werden die UARTs miteinander verbunden? Betrachten wir zunächst eine Konfiguration bestehend aus zwei UARTs. Siehe Bild 3:

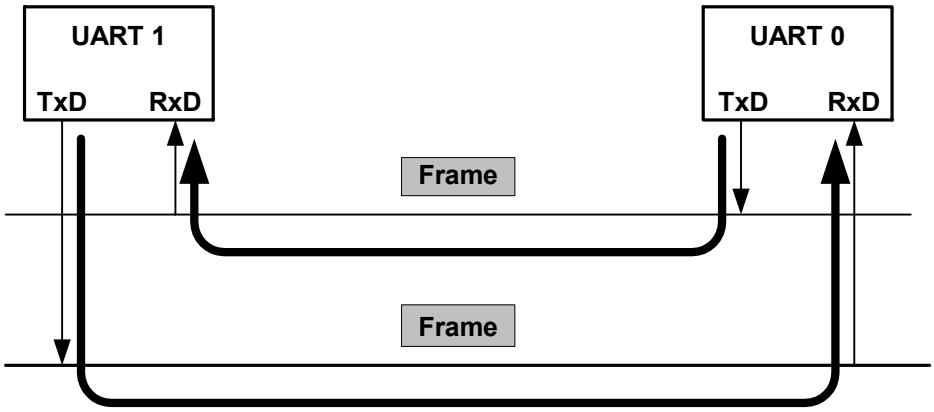


Bild 3: Verbindung von zwei UARTs

Jeder UART besitzt einen TxD-Ausgang (Transmit Data) und einen RxD-Eingang (Receive Data). Wie im Bild zu sehen ist, ist TxD von UART 1 mit RxD von UART 0 verbunden, und umgekehrt TxD von UART 0 ist mit RxD von UART 1 verbunden. Die Kommunikation zwischen den UARTs erfolgt über kleine serielle Dateneinheiten, die als **Frames** bezeichnet werden. Abhängig vom Typ des UARTs bestehen diese Frames aus 1 oder mehreren Bytes, die auch als **Oktets** bekannt sind.
Hat z. B. UART 0 eine Information zur Verfügung, dann sendet er sie in Form eines Frames über TxD und der ersten Verbindungsleitung zum RxD-Eingang von UART 1. Weil zusätzlich eine zweite Verbindungsleitung zur Verfügung steht, kann UART 1 zur gleichen Zeit einen Frame an UART 0 schicken. Weil der Datentransfer gleichzeitig in beide Richtungen erfolgen kann, nennt man diese Art des Datenaustauschs **Vollduplex**-Kommunikation.

Ist die Vollduplex-Kommunikation mit dem bisherigen Verbindungsschema (bekannt als RS232) auch mit mehr als zwei Rechnern möglich? Machen wir einen Versuch und verbinden drei Rechner miteinander. Siehe Bild 4:

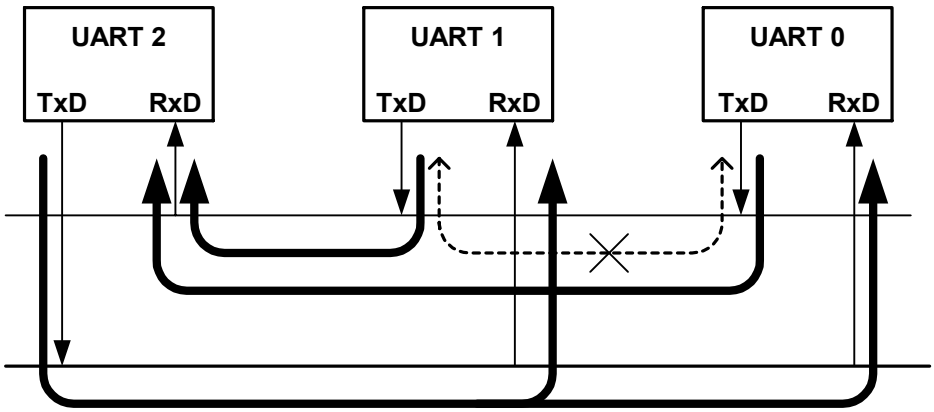


Bild 4: Vollduplex-Kommunikation mit RS232

Es läßt sich erkennen, daß zwischen UART 0 und UART 1 **keine** TxD/RxD-Verbindung besteht. Stattdessen sind ihre TxD-Ausgänge kurzgeschlossen. So kann UART 0 keinen Frame an UART 1 senden und umgekehrt UART 1 kann keinen Frame an UART 0 senden. Mit anderen Worten, sie können nicht miteinander kommunizieren. Eine Vollduplex-Verbindung besteht lediglich zu UART 2. Vergleiche auch mit folgender Tabelle:

		Empfangen (RxD)		
		UART 0	UART 1	UART 2
Senden (TxD)	UART 0		Nein	Ja
	UART 1	Nein		Ja
	UART 2	Ja	Ja	

Was ist zu tun? Die Lösung ist einfach: Verbinde die UARTs nicht mehr mit zwei Leitungen, sondern nur mit einer einzelnen, und schalte zwischen diese Leitung und die RxD/TxD-Anschlüsse der UARTs je einen Tranceiver. Bei diesem Verbindungsschema, das auch als RS485 bekannt ist, bestimmt die DTR-Leitung (Data Terminal Ready) der UARTs die Transferrichtung ihrer Tranceiver.

So ist z.B. mit einem H-Pegel der TxD-Ausgang freigegeben und der RxD-Eingang gesperrt. Der UART kann **senden**. Mit einem L-Pegel ist der RxD-Eingang freigegeben und der TxD-Ausgang gesperrt. Der UART kann **empfangen**.
Die einzelne Leitung des Netzwerks verbindet alle UARTs miteinander, und jeder kann sowohl empfangen als auch senden. Somit kann jeder Rechner mit jedem kommunizieren. Doch die Lösung enthält eine wesentliche Einschränkung. Der Datentransfer in beide Richtungen darf **niemals** gleichzeitig erfolgen. So sollte z.B. nur folgendes geschehen, siehe Bild 5: UART2 sendet zuerst einen Frame an UART 0 (1) und anschließend, wenn die Leitung wieder frei ist, sendet UART 0 einen Frame zurück an UART 2 (2). Diese Art des Datentransfers bezeichnet man als **Halbduplex-**Kommunikation.

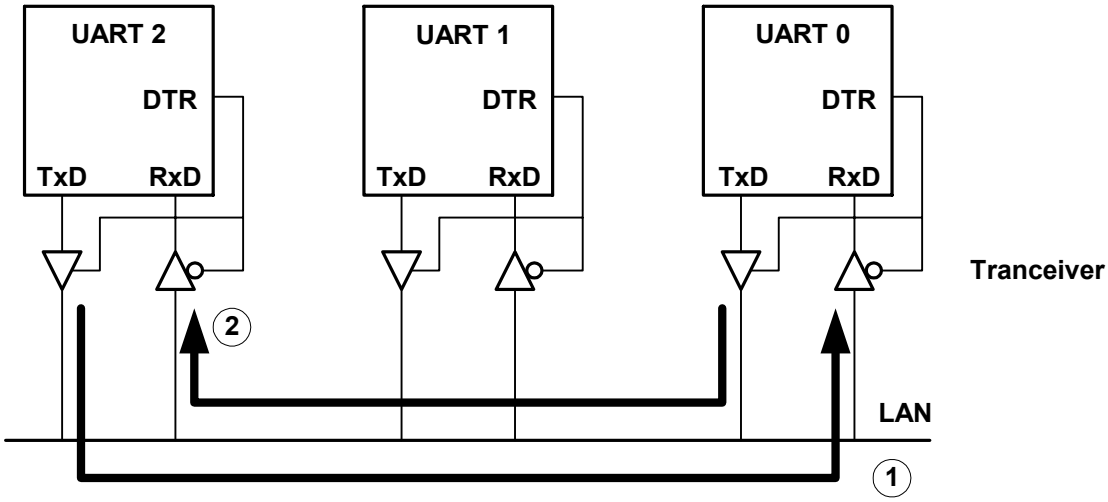


Bild 5: Halbduplex-Kommunikation mit RS485

Wenn ein Frame von einem Rechner zu einem anderen übertragen wird, dann ist dafür eine bestimmte Zeit erforderlich. Diese sogenannte **Framezeit** t_f errechnet sich wie folgt:

$$t_f = \textit{Anzahl der Bits je Frame dividiert durch die Baudrate}$$

Angenommen, ein Frame besteht aus 14 Oktets zu je 8 Bits und wird asynchron (Start- und Stopp-Bit begrenzen jedes Oktet) übertragen. So besteht jedes Oktet aus 10 Bits und der Frame aus insgesamt 10 Bits · 14 Oktets = 140 Bits. Ist die Baudrate z.B. 9600 Bits/s, dann beträgt die Framezeit t_f = 140/9600s = 0,0146s = **14,6 ms**. Während dieser Zeit befindet sich der Frame im Netzwerk. Siehe Bild 6:

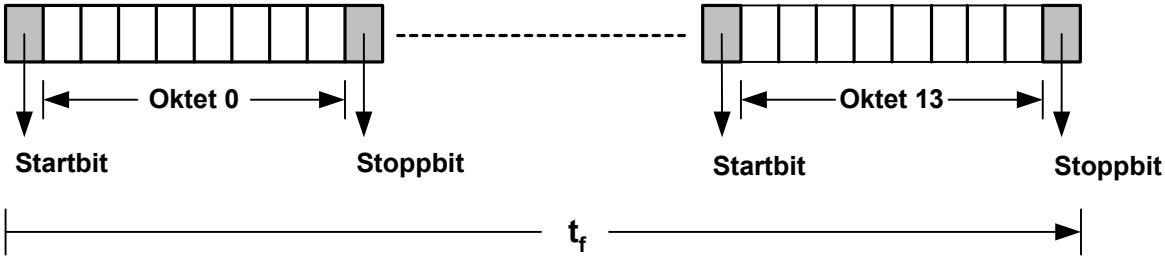


Bild 6: Frame während t_f im Netzwerk

Bei der Halbduplex-Kommunikation kommt ein Frame nur dann unbeschadet beim Empfänger an, wenn während der Framezeit kein anderer UART sendet. Sollte jedoch dieses Zeitraster nicht garantiert sein, dann kommt es zur Kollision der Frames. Betrachten wir dazu folgendes Szenarium: UART 0 schickt zum Zeitpunkt t_0 einen Frame in das Netzwerk. Zum Zeitpunkt t_0 + t sind Oktet 0, 1 und 2 bereits beim Empfänger angekommen, und Oktet 3 befindet sich gerade auf der Strecke. Genau zu diesem Zeitpunkt beginnt UART 1 zu senden und schickt sein Oktet 0 in das Netzwerk. Siehe Bild 7. Bit für Bit kollidieren nun die beiden Oktets, und das gleiche geschieht mit den Nachfolgern. Kein Frame kommt unbeschädigt bei den Empfängern an.

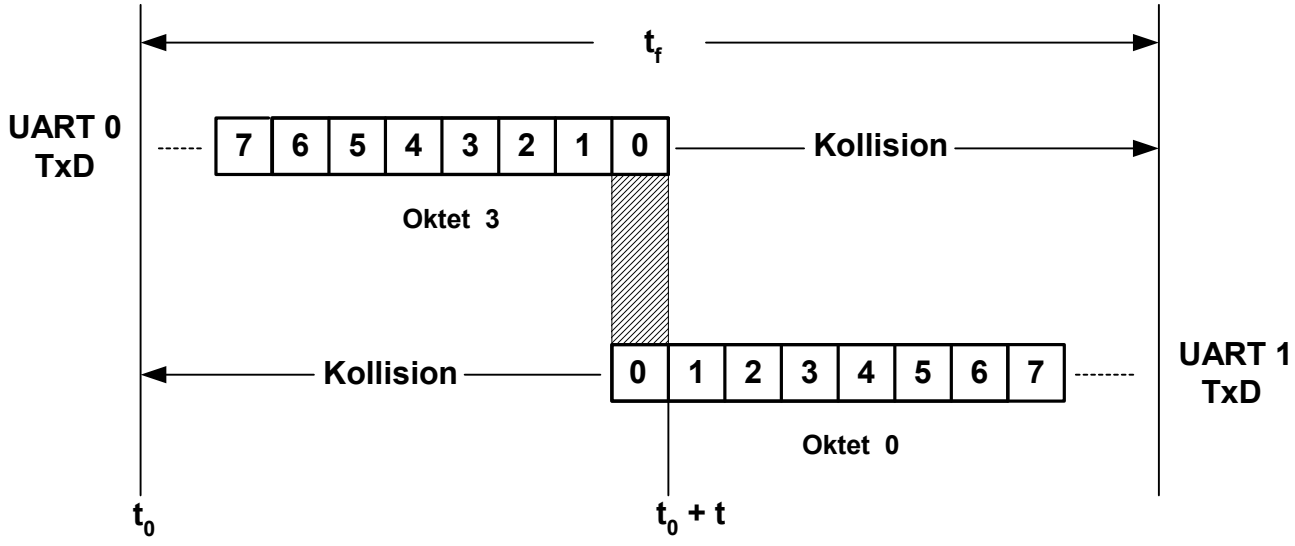


Bild 7: Kollision während der Framezeit

1.2 Kollisionsfreie Übertragung

Die Kollisionen im LAN können nur vermieden werden, wenn aus der Menge von Stationen (Rechnern) nur eine einzelne Station für eine begrenzte Zeit Sendeberechtigung hat, während die verbleibenden Stationen in dieser Zeit im Empfangsbetrieb bleiben.
Eine Realisierung dieses Verfahrens ist das **tokenbus-basierte** LAN, das speziell in der industriellen Produktion (Fabrikautomatisierung) eingesetzt ist. Im Bürobereich ist stattdessen das **CSMA/CD**-Verfahren (Carrier Sense Multiple Access with Collision Detection; Trägererkennung mit Mehrfachzugriff und Kollisionserkennung) im Einsatz.

1.2.1 Tokenbus-Operationen

Die Tokenbus-Operationen basieren auf einem Protokoll, das für die kollisionsfreie Realisierung der Datenübertragung einen speziellen Frame benutzt. Er besteht aus mehreren Oktets, und eines davon trägt den Wert **08** Hex. Dieser Wert spezifiziert das sogenannte **Token**. Daher wird dieser Frame auch als **Token Control Frame** bezeichnet. Diejenige Station, die im Besitz des Tokens ist, hat Sendeberechtigung.
Jede Station ist durch eine individuelle Adresse im Bereich von **1..n** identifiziert. Die Adresse 0 kommt nicht vor. Der Token Control Frame hat damit die Möglichkeit das Token von der höchsten Stationsadresse zur niedrigsten Stationsadresse zu transportieren. Mit anderen Worten, das Token wandert von der Vorgängerstation (PS=Previous Station) über die augenblickliche Station (TS=This Station) zur Nachfolgestation (NS=Next Station) in einem logischen Ring. Dabei darf nur der Besitzer des Tokens den Control Frame weiterleiten. Siehe Bild 8:

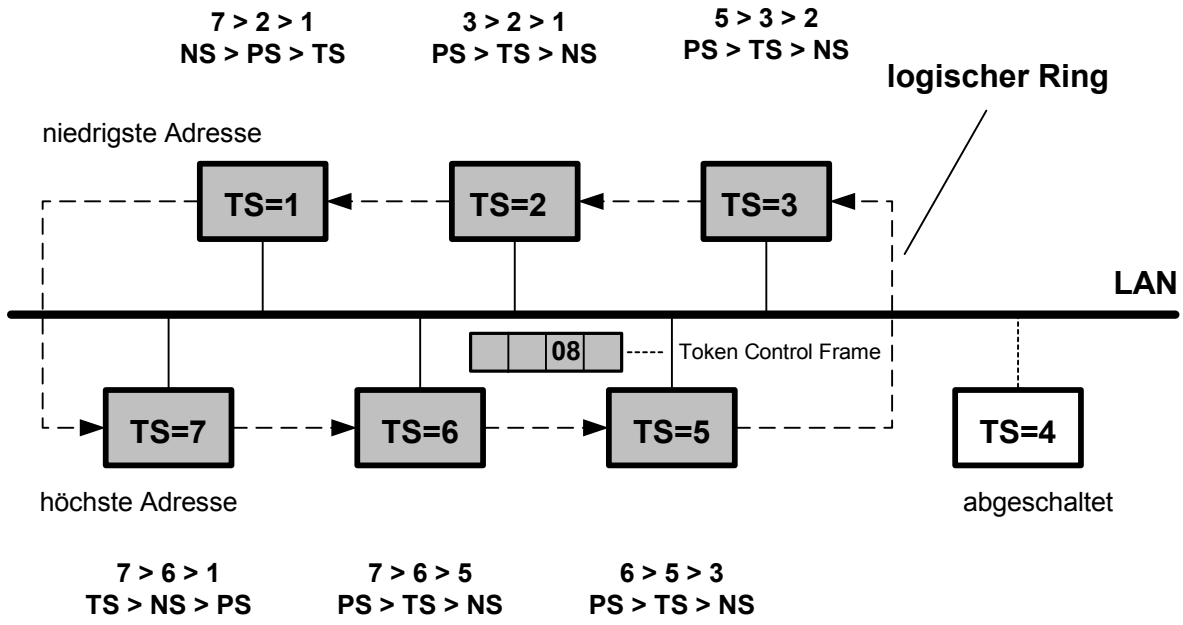


Bild 8: Token-Bus

Wie zu erkennen ist, wird das Token und damit das Recht zum Senden von Station zu Station in absteigender numerischer Reihenfolge weitergegeben. Ist das Token bei der niedrigsten Adresse angekommen, wird es an die Station mit der höchsten Adresse übertragen. Hier schließt sich der logische Ring.

Es spielt keine Rolle, in welcher physikalischen Reihenfolge sich die Stationen im Ring befinden. Auch müssen nicht alle Stationen zugeschaltet sein. So ist z. B. Station 4 abgeschaltet. Daher gibt Station 5 das Token weiter an Station 3. Wenn eine Station das Token hat, kann sie eine bestimmte Zeitspanne Daten senden. Danach gibt sie das Token weiter zur nächsten Station. Hat eine Station das Token erhalten, und es stehen keine Sendedaten bereit, gibt sie das Token sofort nach ihrem Erhalt weiter.

1.3 Die interne Struktur der MAC-Teilschicht

Die Beschreibungen und Spezifikationen der MAC-Teilschicht basieren auf den Empfehlungen des IEEE 802.4-Komitees. Betrachten wir zunächst ihre interne Struktur. Sie ist im Bild 9 illustriert:

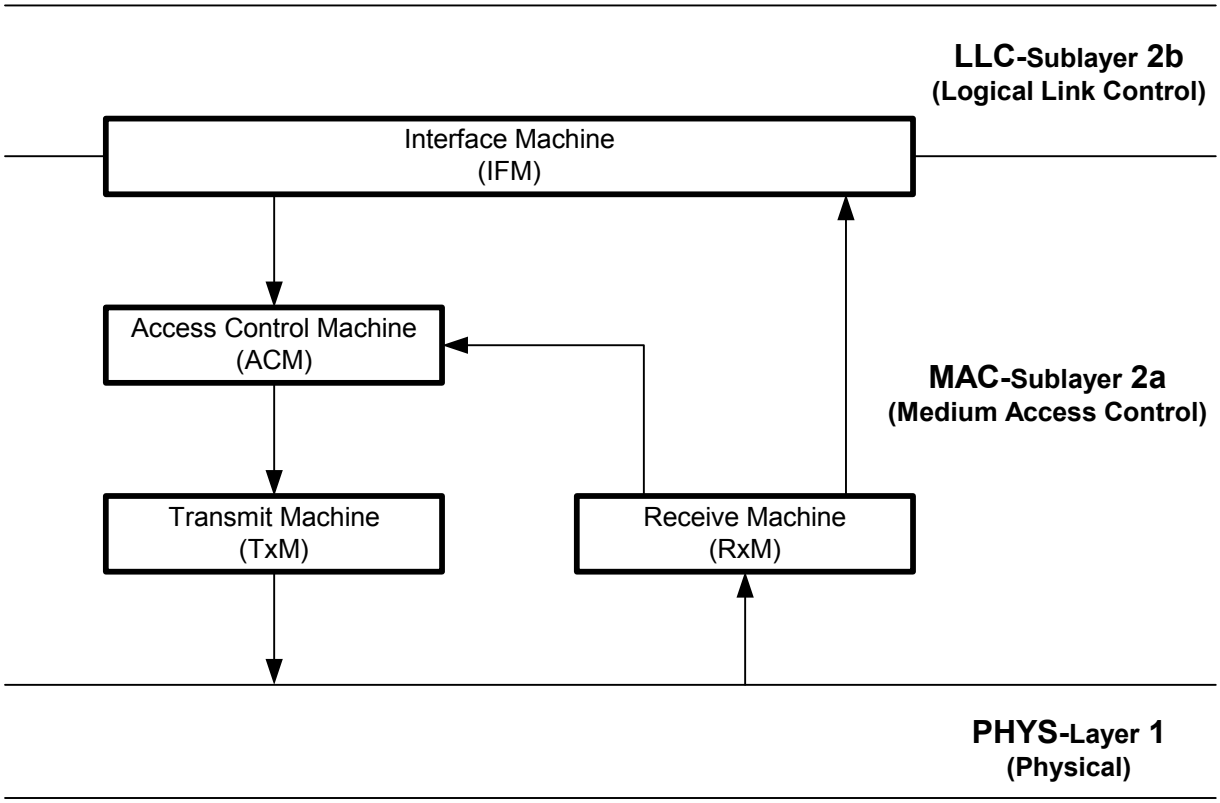


Bild 9: Interne Struktur der MAC-Teilschicht

Die Struktur zeigt eine Aufteilung in vier verschiedene Bereiche, die als logische ´Maschinen´ aufgefaßt werden können. Jede von ihnen übernimmt spezielle MAC-Funktionen:

a**, Die Interface Machine (IFM)**
Sie arbeitet als Schnittstelle und Puffer zwischen der Logical Link Control (LLC)-Teilschicht 2a und der Medium Access Control (MAC)-Teilschicht 2b.

b, **Die Access Control Machine (ACM)**
Sie ist das Herzstück des Tokenbus-Systems. Sie bestimmt, wann ein Frame in das Netz geschickt wird. Zu diesem Zweck kooperiert sie mit den ACMs aller anderen Stationen.

c, **Die Receive Machine (RxM)**
Sie korrespondiert mit dem Physical Layer und bekommt von dort MAC Frames bzw. MAC **P**rotocol **D**ata **U**nits (PDUs). Sie überprüft die **F**rame **C**heck **S**equence (FCS) der einzelnen PDUs, stellt Fehlermeldungen des UART fest und gibt die Ergebnisse weiter an die Interface Machine und die Access Control Machine.

d, **Die Transmit Machine (TxM)**
Sie bekommt Daten-Frames von der Access Control Machine, berechnet von jedem die Frame Check Sequence, hängt sie an die Daten-Frames an und schickt sie als MAC PDUs an den Physical Layer.

1.3.1 Die Receive Machine

Die Receive Machine hält sich wie alle anderen MAC-Machines in der Privileg-Ebene 0 auf. Sie ist als Interrupt-Task implementiert und unterliegt deshalb ausschließlich dem Prioritäts-Schedulung (Prioritätsstufe 3). Identifiziert ist sie durch die folgenden beiden Datenstrukturen:

- Task-Status-Segment (TSS) **RxM_Task** und
- Task-Control-Block (TCB) **task_cb[10]**.

RxM_Task-Control Block

naechste_task
task_nummer = 10
task_status = 1
task_prioritaet = 3

task_cb[10]

Task-Status-Segment der **RxM_Task**

```
RxM_Task (DPL=0, PRESENT
          Code = start_10_proc,
          Data = RxM.Data,
          STACKS = (RxM.STACK,
                    OS_Kern.STACK),
          LDT = RxM_Task_LDT,
          IOPRIVILEGE = 0,
          INTENABLE);
```

Build386-Spezifikationen

Bild 9a: Identifikation der RxM_Task

Auf die Build386-Spezifikationen (siehe dazu Teil 1: Memory Management und System Design im Protected Mode der x86/Pentium-Architektur) und die Initialisierung des Task-Control-Blocks, die von einer speziellen Betriebssystem-Kernfunktion durchgeführt wird (siehe dazu Teil 2: Der Kernel), soll an dieser Stelle nicht eingegangen werden.

Gestartet wird die Receive Machine über ihre zugängliche Funktion **RxM_exe()**, die von der Privileg-Ebene 3 aus über den Call Gate-Deskriptor **KN_rxm_exe** und den Assembler-Adapter **adapt019** erreichbar ist. Siehe Bild 10:

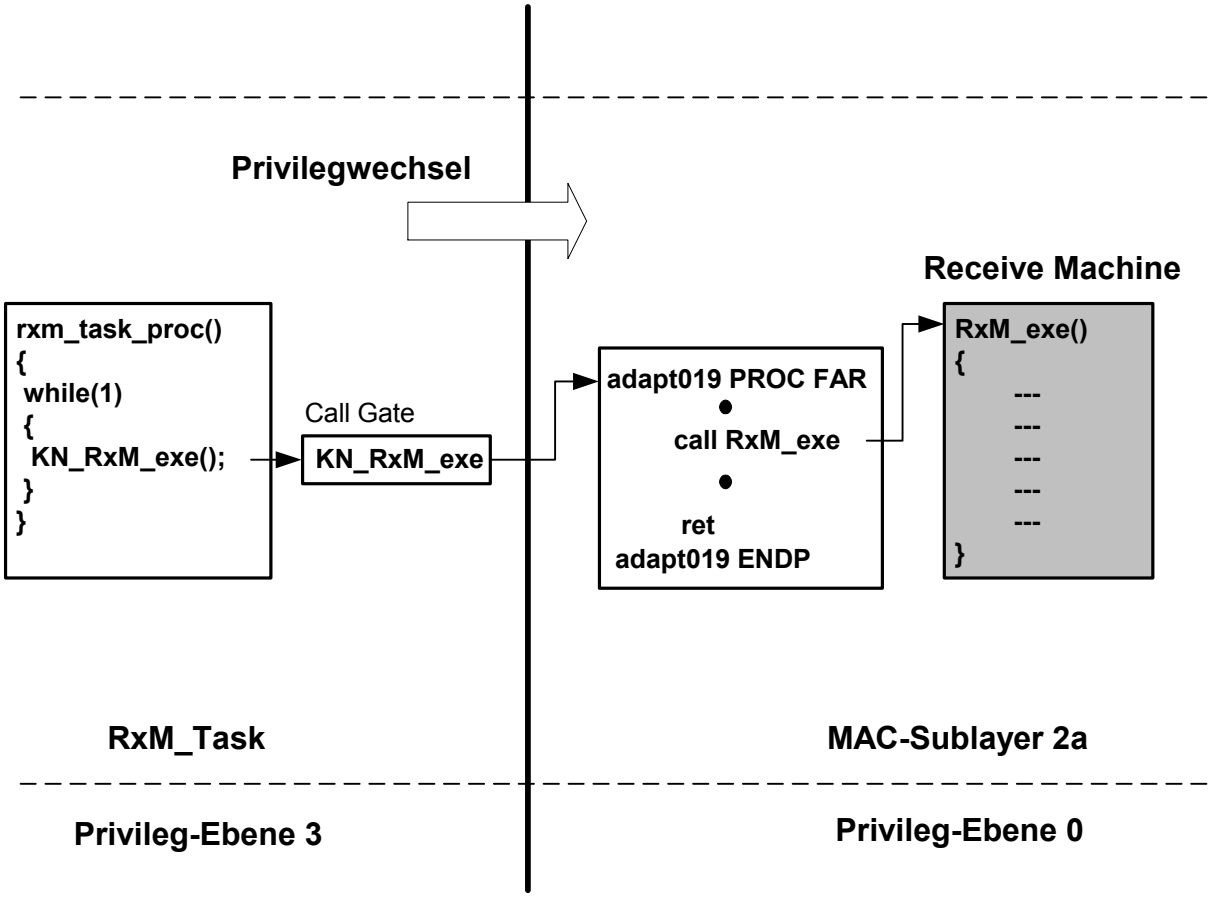


Bild 10: Starten der Receive Machine

Wie zu sehen ist, befindet sich die RxM_Task im **MAC-Sublayer 2a**. Nach dem Wechsel in die Privileg-Ebene 0 führt sie dort in einer Endlosschleife die Betriebssystem-Funktion RxM_exe() aus und kehrt nie wieder in die Privileg-Ebene 3 zurück. Die vollständige Funktion zeigt Bild 11:

```
include<i86.h>

#define warten_bis_interrupt29_eintrifft        29
#define rxm_max                                 14
#define token                                 0x08
#define data                                  0x40
#define io_error                              -1
#define mask                                0x1021
#define ctrl                                    2
#define rbr1                                0x380
#define lsr1                                0x385
#define iir1                                0x382
#define fcr1                                0x382
#define ok                                      0

extern unsigned short int crc(unsigned short int,unsigned short int,
                              unsigned short int);
static unsigned short int  fcs;
static unsigned char       i;
static char                acm_event;
static char                llc_event;

unsigned char              rxm_puffer[rxm_max] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0};

void RxM_exe()
{
 while(1)
 {
  wait(warten_bis_interrupt29_eintrifft);

  switch(inbyte(iir1))
  {
                                                        /* Frame verfuegbar */
   case 0xc4:  i=0;
          for(i=0;i<rxm_max;i++)
             rxm_puffer[i] = inbyte(rbr1);

          outbyte(fcr1,0xcb);                          /*Receiver-FIFO Reset */

          fcs=0;                                         /*   fcs berechnen    */
          for(i=0;i<rxm_max;i++)
             fcs=crc(fcs,rxm_puffer[i],mask);

          switch(fcs)
          {
           case 0:
            switch(rxm_puffer[ctrl] & 0x48)
            {
             case data:                                  /*    data empfangen   */
                    llc_event=ok;
                    sende_llc_event(llc_event,rxm_puffer);          /* an ifm */

                    acm_event=ok;
                    sende_acm_event(acm_event,rxm_puffer);        /* an acm */
              break;
```

```
            case token:                                    /*   token empfangen  */
              acm_event=ok;
              sende_acm_event(acm_event,rxm_puffer);              /* an acm */
            break;

            case claim:                                    /*   claim empfangen  */
              acm_event=ok;
              sende_acm_event(acm_event,rxm_puffer);       /*     an acm     */
             break;
            }
           break;
           default:                                          /*   fcs-Error     */
            acm_event=io_error;
            sende_acm_event(acm_event,rxm_puffer);         /*     an acm      */
           break;
          }
   break;
                                                        /*  Ov-, Par-, Fra-Error  */
   case 0xc6: inbyte(lsr1);                           /* clear line status-interrupt */
          outbyte(fcr1,0xcb);                           /*Receiver-FIFO Reset */

          acm_event = io_error;
          sende_acm_event(acm_event,rxm_puffer);              /*     an acm      */
   break;
                                                        /*  Character timeout    */

   case 0xcc: inbyte(rbr1);                           /* clear timeout-interrupt */
          outbyte(fcr1,0xcb);                           /* Receiver-FIFO Reset */

          acm_event = io_error;
          sende_acm_event(acm_event,rxm_puffer);              /*     an acm      */
   break;
  }
 }
}
```

Bild 11: Betriebssystem-Funktion RxM_exe (Datei rxmexe.c)

Die Anweisungen der Receive Machine sollen nun im einzelnen besprochen werden. Zunächst sei angemerkt, daß die Receive Machine sehr eng mit dem UART zusammenarbeitet. Von ihm bekommt sie die fehlerfrei übertragenen MAC PDUs und Meldungen von Übertragungsfehlern, die der UART mindestens aufspüren kann. Betrachten wir zunächst das allgemeine Format und die Komponenten der MAC PDU nach IEEE 802.4. Siehe Bild 12:

Bild 12: Allgemeines Format der MAC PDU

Präambel = Ein Bitmuster (1 oder mehrere Oktets), das jedem übertragenen Frame vorausgeht.
SD = Start Delimiter (1 Oktet). Ein Bitmuster, das den Beginn eines Frames kennzeichnet.
FC = Frame Control (1 Oktet). Bestimmt, welche Klasse von Frame übertragen wird: **Data Frame** oder **MAC Control Frame** (z.B. token).
DA = Destination Address (2 oder 6 Oktets). Enthält die Adresse derjenigen Station, an die ein Daten- oder MAC Control Frame geschickt wird.
SA = Source Address (2 oder 6 Oktets). Enthält die Adresse derjenigen Station, die einen Daten- oder MAC Control Frame abschickt.
Daten = Informationen des nächsthöheren LLC-(Logical Link Control) Sublayers 2b (0 oder mehrere Oktets).
FCS = Frame Check Sequence (4 Oktets). Eine ausführliche Erklärung folgt etwas später.
ED = End Delimiter (1 Oktet). Ein Bitmuster, das das Ende eines Frames kennzeichnet.

Wenn zwischen der Receive Machine und dem UART ein Interrupt-Korrespondenzschema besteht, dann können die Komponenten der MAC PDU auf ein Minimum reduziert werden. Von Nutzen ist dabei ein UART, der zur Aufnahme eines kompletten Frames mit einem FIFO ausgestattet ist. Ein solcher UART ist z.B. der TL16C552. Sein FIFO kann maximal 16 Oktets aufnehmen, und er läßt sich so programmieren, daß er eine Interruptanforderung absetzt, wenn in seinem FIFO 1, 4, 8 oder 14 Oktets eingetroffen sind. Dieser sogenannte Interrupt Trigger Level fordert die Receive Machine auf, in einer for-Schleife die entsprechende Anzahl von Oktets zu holen, um sie anschließend als MAC PDU zu interpretieren. So signalisiert der Interrupt Trigger Level sowohl den Anfang als auch das Ende des eingetroffenen Frames. Die Präambel, der Start- und der End Delimiter können deshalb entfallen.
im Programm-Listing (Bild 11) ist zu sehen, daß die Receive Machine als erstes

wait(warten_bis_interrupt29_eintrifft);

ausführt. Die RxM_Task geht daraufhin in den Wartezustand und wird aus diesem Zustand durch ein entsprechendes signal befreit, das der UART über den Interrupt Trigger Level auslöst. Auf welche Weise dies geschieht, soll nun erläutert werden. Betrachten wir als erstes das Interrupt-Anforderungsschema des UART. Siehe Bild 13:

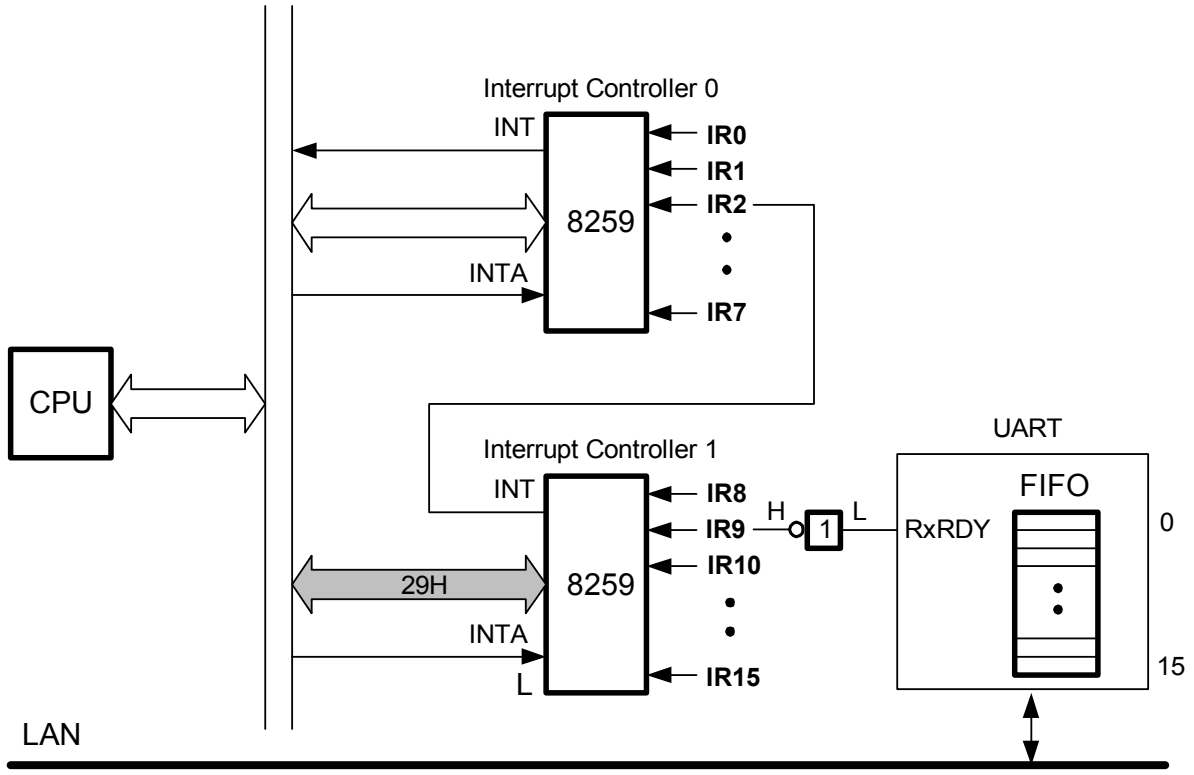


Bild 13: Interrupt-Anforderungsschema des UART

Wie zu erkennen ist, sind die beiden Interrupt-Controller als Kaskade konfiguriert (wie auf einem PC-Motherboard). Der Interrupt-Controller 0 arbeitet als Master und der Interrupt-Controller 1 als Slave. Sein INT-Ausgang ist mit dem Interrupt Request-Eingang IR2 des Masters verbunden. Der RxRDY-Ausgang (Receiver Ready) des UART ist über einen Inverter am Interrupt Request-Eingang IR9 des Slave angeschlossen. Wenn der UART den programmierbaren Interrupt Trigger Level erreicht hat, wir wählen 14 Oktets, dann meldet er dieses Ereignis mit RxRDY = Low-Pegel. Der Inverter invertiert diesen Pegel und gibt ihn als Interrupt-Anforderung weiter an den IR9-Eingang des Slave. Dieser gibt ihn weiter an den Master (IR2-Eingang) und dieser schließlich an die CPU. Die CPU quittiert die Anforderung mit INTA = Low-Pegel (Interrupt Acknowledge) und liest damit das ICW1-Register (Initial Code Word) des Slave. Dieses Register ist so programmiert, daß der Prozessor aufgrund einer Anforderung an IR8 mit INTA über den Datenbus das Vektorbyte 28H liest. Bei einer Interruptanforderung an IR9 bekommt er deshalb die nächsthöhere Nummer, also das Vektorbyte **29H** und interpretiert diesen Wert als Index bzw. Offset eines Interrupt Gate-Deskriptors in der IDT (Interrupt Deskriptor Tabelle). Siehe Bild 14:

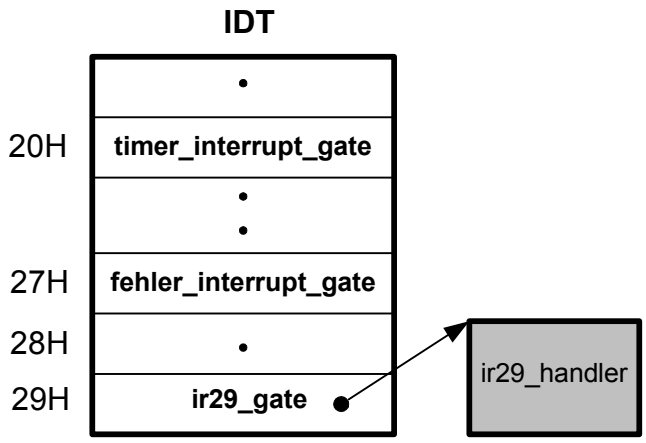


Bild 14: Layout der Interrupt-Deskriptor-Tabelle

Der Name des Interrupt-Gate-Deskriptors lautet **ir29_gate** und er zeigt auf die Assembler-Interrupt-Prozedur **ir29_handler**. Sie ist Mitglied des Betriebssystem-Kerns und rettet als erstes alle Register und Selektoren der unterbrochenen Task. Sie setzt das NT-Bit = 1, um einen potentiellen Taskwechsel zu ermöglichen, lädt den DS-Selektor der OS_Kern_Task und ruft schließlich die Betriebssystem-Funktion **signal_ir29** auf. Danach sorgt sie für die Rückkehr zur unterbrochenen Task in der Privileg-Ebene 3. Sie löscht deshalb das NT-Bit und stellt damit den alten Zustand wieder her, lädt die Register und Selektoren der unterbrochenen Task zurück und führt schließlich wegen NT = 0 den Befehl iretd erfolgreich aus. Siehe Bild 15:

```
        NAME ir29hand


PUBLIC ir29_handler
EXTRN  signal_ir29:NEAR

DATA SEGMENT RW PUBLIC
   DD  ?
DATA ENDS

CODE32   SEGMENT ER PUBLIC

ir29_handler PROC FAR WC(0)
     pushad                        ;Alle Register und alle Selektoren der
     push ds                       ;unterbrochenen Task retten
     push es
     push fs
     push gs

     pushfd                        ;Taskwechsel ermoeglichen
     pop  eax
     or   ax,0100000000000000B  ;NT-Bit=1 setzen (wegen Task-Verkettung)
     push eax
     popfd

     mov  ax,data                  ;DS-Selektor der OS-Kern-Task laden
     mov  ds,ax
     call   signal_ir29

     pushfd                        ;Rueckkehr zur unterbrochenen
     pop  eax                      ;Anwender-Task ermoeglichen
     and  ax,1011111111111111B ;NT-Bit=0 setzen (alter Zustand)
     push eax
     popfd

     pop  gs                       ;Alle Register und alle Selektoren
     pop  fs                       ;der unterbrochenen Task zurueckladen
     pop  es
     pop  ds
     popad
     iretd                         ;Zurueck zur unterbrochenen Task
ir29_handler ENDP

CODE32   ENDS
      END
```

Bild 15: Betriebssystem-Prozedur ir29_handler (Datei ir29hand.src)

Die Betriebssystem-Funktion signal_ir29 signalisiert der wartenden RxM_Task, daß ein Frame bzw. eine MAC PDU im FIFO des UART zur Abholung bereit steht. Sie verläßt daraufhin ihren Wartezustand und wird anschließend als ready-Task an der höchsten Prioritätsstufe gestartet. Siehe Bild 16:

```
#define warten_bis_interrupt29_eintrifft   29

void signal_ir29(void)
{
 signal(warten_bis_interrupt29_eintrifft);
}
```

Bild 16: Betriebssystem-Funktion signal_ir29 (Datei sigir29.c)

1.3.1.1 Hardware-basierte Fehlererkennung

Die RxM_Task setzt die Ausführung der Receive Machine fort, und überprüft als erstes das Interrupt Identifikation Register **iir1** des UART. In einer switch-Anweisung wertet sie drei verschiedene Codes aus, und stellt dabei fest, ob der Frame unbeschädigt angekommen ist oder nicht.

```
switch(inbyte(iir1))
{
 case 0xc4:                    /* Frame verfügbar */
            -----
            -----
 break;
 case 0xc6:                  /* Ov-, Par-, Fra-Error */
            -----
            -----
 case 0xcc:                   /* Character timeout */
            -----
            -----
 break;
}
```

Der Code 0xc4 zeigt an, daß der Frame fehlerfrei angekommen ist. Die restlichen zwei Codes identifizieren Übertragungsfehler im LAN. Mehr Fehler kann der benutzte UART nicht aufspüren. Betrachten wir zunächst den Code 0xc6. Er meldet Overrun (Ov)-, Parity (Par)- und Frame (Fra)-Error:

- Ein **Overrun-Error** tritt auf, wenn ein eingetroffener Frame nicht abgeholt und von einem nachfolgenden Frame überschrieben worden ist (Frame-Verlust).

- Ein **Parity-Error** tritt auf, wenn der Empfänger in mindestens einem Oktet des Frames eine falsche Parität entdeckt. Wie ist darunter zu verstehen? Betrachten wir dazu ein Beispiel. Der UART auf der Senderseite bestimmt für jedes Oktet ein Paritätsbit und fügt es an das originale Oktet an. Wie dies geschieht zeigt das folgende Bild 17:

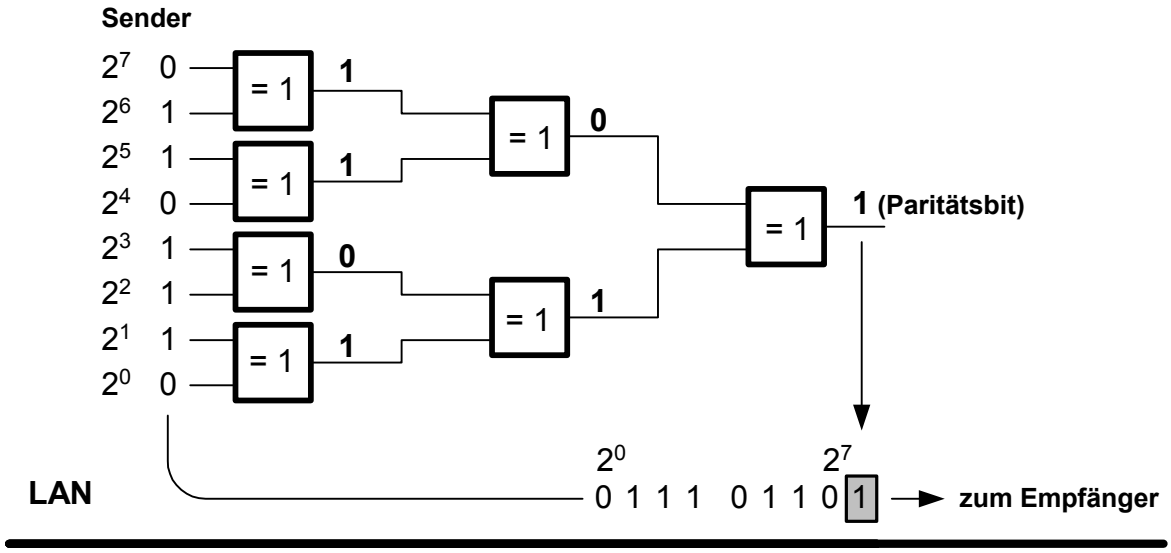


Bild 17: Erzeugung des Paritätsbits auf der Senderseite

Der UART sei so programmiert, daß er den Bitstrom des originalen Oktets auf gerade Parität ergänzt. Ist der Bitstrom des Oktets z.B. (LSB)**0111 0110**(MSB), dann ist die Anzahl der 1en ungerade, nämlich 5. Der Sender muß den Bitstrom um das Paritätsbit = 1 ergänzen. Damit wird die Parität des resultierenden Bitstroms gerade, nämlich 6. Im Bild 17 ist zu erkennen, daß der Sender das Paritätsbit auf einfache Weise mit Hilfe einer dreistufigen EXOR-Verknüpfung erzeugen kann.
Wenn auf dem Weg zum Empfänger eine Anzahl von 1en im Bitstrom zerstört wird, dann kommt das Oktet beschädigt beim Empfänger an. Was macht der Empfänger mit dem erhaltenen Bitstrom? Er spaltet das empfangene Paritätsbit ab, erzeugt aus dem originalen Oktet wiederum ein Paritätsbit und vergleicht das selbst erzeugte und empfangene Paritätsbit miteinander. Bei Gleichheit scheint das Oktet unbeschädigt angekommen zu sein, bei Ungleichheit nicht. Angenommen Bit 2^6 im Originalstrom wurde zerstört, so wird aus (LSB)**0111 0110**(MSB)1(Par.) der Bitstrom (LSB)**0111 0100**(MSB)1(Par.). Der Empfänger erzeugt Paritätsbit = 0 und stellt zwischen erzeugtem und empfangenem Paritätsbit Ungleichheit fest. Siehe Bild 18:

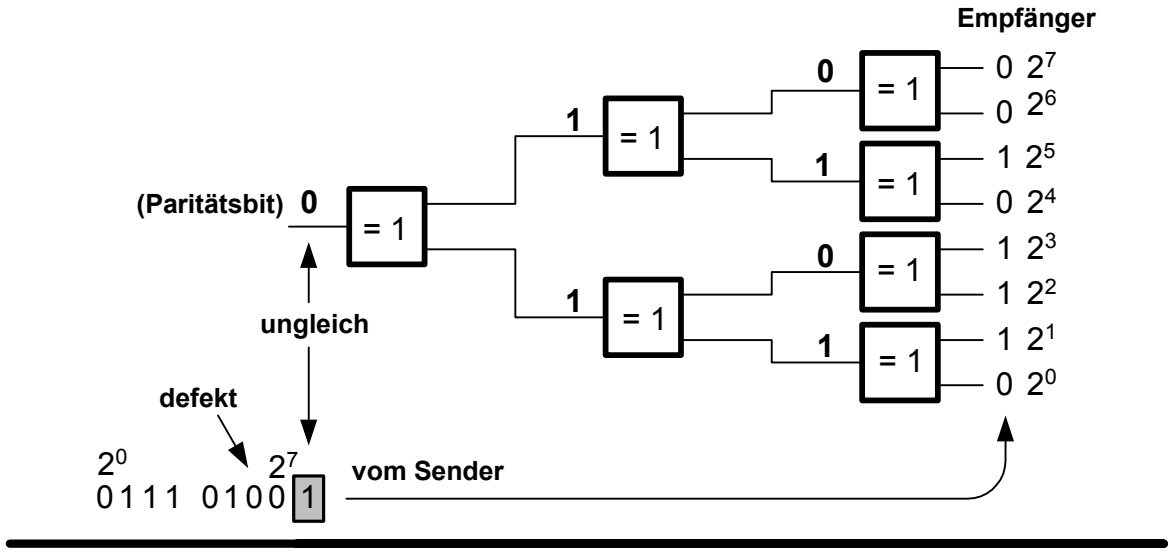


Bild 18: Erzeugung des Paritätsbits auf der Empfängerseite und Vergleich

Doch die Prioritätsüberwachungsmethode hat Schwächen. Sie sind in der folgenden Tabelle verdeutlicht:

Sender			**Empfänger**		
Anzahl der Bits im originalen Bitstrom	**defekte Bits im originalen Bitstrom**	**übertragenes Paritätsbit**	**ankommender originaler Bitstrom**	**erzeugtes Paritätsbit**	**Fehler-erkennung**
ungerade (0111 0110)	**gerade (2 Bits)**	**unbeschädigt (1)**	**ungerade (0101 0010)**	**1**	**nein**
ungerade (0111 0110)	**ungerade (3 Bits)**	**unbeschädigt (1)**	**gerade (0000 0110)**	**0**	**ja**
ungerade (0111 0110)	**gerade (2 Bits)**	**beschädigt (0)**	**ungerade (0101 0010)**	**1**	**ja**
ungerade (0111 0110)	**ungerade (3 Bits)**	**beschädigt (0)**	**gerade (0000 0110)**	**0**	**nein**

Bei der Übertragung des Bitstroms vom Sender zum Empfänger kann eine gerade Anzahl oder eine ungerade Anzahl von Bits im im originalen Bitstrom beschädigt werden. Aber auch das übertragene Paritätsbit selbst kann defekt beim Empfänger ankommen.
Wie die Tabelle zeigt, gibt es vier verschiedene Fehlersituationen, von denen die Prioritätsüberwachungsmethode jedoch nur **zwei** erkennen kann. Was ist zu tun?
Eine leistungsfähigere Fehlererkennungsmethode (Stichwort CRC; Cyclic Redundancy Code) könnte die Prioritätsüberwachungsmethode ablösen. Doch wir werden bald einsehen, CRC ist zwar der Prioritätsüberwachungsmethode weit überlegen, eine 100%ige Fehlererkennung garantiert aber auch CRC nicht. Und so ist es besser mehrere Methoden zu kombinieren.

- Ein **Frame-Error** tritt auf, wenn der Empfänger am Ende eines Oktets das Stoppbit abtastet und statt eines High-Pegels einen Low-Pegel identifiziert. Wie kann es zu einer solchen Situation kommen? Betrachten wir dazu Bild 19:

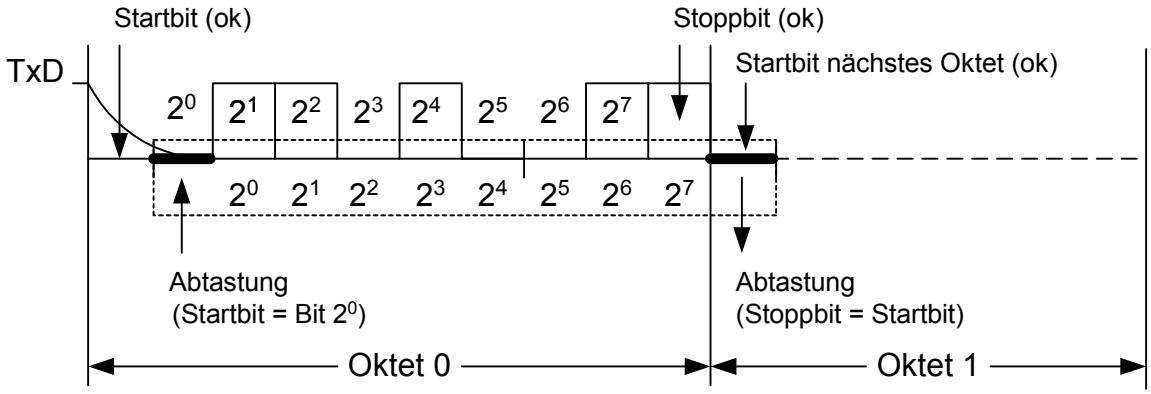


Bild 19: Frame Error Erkennung

Wenn der Sender-UART ein neues Oktet in das LAN schickt, dann schaltet er den TxD-Ausgang (Transmit Data) von High-Pegel auf Low-Pegel und hält diesen Pegel für die Dauer eines Bits. Der Empfänger-UART tastet diesen Pegel ab und interpretiert ihn als Startbit (Zeichensynchronisation). Am Ende des Oktets fügt der Sender-UART für die Dauer eines Bits einen High-Pegel hinzu, den der Empfänger-UART als Stoppbit interpretiert. Sollte der Empfänger-UART das Startbit **nicht** erkennen, weil sein Low-Pegel noch nicht erreicht ist, dann interpretiert er den nächstmöglichen Low-Pegel als Startbit. In Bild 19 ist dies z.B. Bit 2^0 des Oktets 0. Auf der Empfängerseite verschiebt sich daher die Zeichensynchronisation um ein Bit nach rechts. Und so interpretiert der Empfänger-UART das Stopbit als Bit 2^7 und das Startbit (**Low**-Pegel) des nächsten Oktets 1 als Stopbit. Der UART meldet Frame Error.
Betrachten wir zuletzt den Code **0xcc**. Er identifiziert einen sogenannten **character timeout**. Er kommt vor, wenn z.B. zwischen zwei empfangenen Oktets eines Frames eine zu große zeitliche Lücke besteht (mehr als 4 zusammenhängende Oktetzeiten) oder wenn zwischen zwei FIFO-Reads eine zu große zeitliche Lücke besteht (ebenfalls mehr als 4 zusammenhängende Oktetszeiten).
Hat der UART keinen der oben beschriebenen Fehler erkannt, dann steht ein verwertbarer Frame zur Verfügung (Code 0xc4 im Interrupt Identifikation Register). In einer for-Schleife entleert die Receive Machine den FIFO und legt die einzelnen Oktets im rxm_puffer[rxm_max] ab. Dieser hat eine Größe von 14 Bytes (#define rxm_max 14). Der Zugriff auf den FIFO erfolgt über inbyte(rbr1), wobei rbr1 die Adresse des FIFO darstellt:

```
for(i=0;i<rxm_max;i++)
        rxm_puffer[i] = inbyte(rbr1);
```

Der FIFO ist als Ringpuffer organisiert und indiziert, beginnend ab Oktet 0, nach jedem Lesezyklus das nächste Oktet im FIFO selbständig.
Unabhängig davon, ob der UART einen Fehler aufgespürt hat oder nicht, mit

```
outbyte(fcr1,0xcb)
```

wird der FIFO in einen definierten Anfangszustand gebracht, und ist damit zur Aufnahme eines neuen Frames vorbereitet. fcr1 stellt die Adresse des FIFO-Control-Registers dar und 0xcb den Reset-Code.

1.3.1.2 Die Komponenten der MAC PDU

Den vom UART empfangenen Frame interpretiert die Receive Machine als MAC PDU (Protocol Data Unit). Sie besteht aus drei Feldern:

- Dem Header, bzw. dem MAC Control-Feld
- dem MAC-Datenfeld und
- dem FCS-Feld (Frame Check Sequence). Siehe Bild 20:

MAC-Control			MAC-Daten									FCS	
Oktet 0	1	2	3	4	5	6	7	8	9	10	11	12	Oktet 13

Bild 20: Layout der empfangenen MAC PDU

Im Vergleich zur allgemeinen Form der MAC PDU nach IEEE 802.4 ist die Anzahl der Oktets auf 14 begrenzt. So besteht z.B. das FCS-Feld nicht aus 4, sondern aus 2 Oktets. Die Präambel, der Start- und der End Delimiter kommen aus den früher beschriebenen Gründen nicht vor. Ansonsten sind in der empfangenen PDU alle empfohlenen Komponenten enthalten. Betrachten wir zunächst den Header der MAC PDU, also das MAC-Control-Feld. Siehe Bild 21:

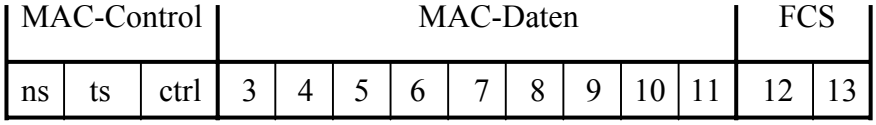


Bild 21: Die Komponenten des MAC-Control-Felds

Es enthält drei Komponenten:

ns = **next station** (1 Byte). Enthält die Adresse derjenigen Station, an die ein Daten-Frame oder ein MAC-Control-Frame geschickt werden soll.

ts: = **this station** (1 Byte). Enthält die Adresse derjenigen Station, von der ein Daten-Frame oder ein MAC-Control-Frame abgeschickt wird.

ctrl: = **control** (1 Byte): Bestimmt welche Klasse von Frame übertragen wird. Eine Auswahl aus der IEEE 802.4-Empfehlung ist:

0x00 (Claim-Frame)
0x08 (Token-Frame)
0x40 (Daten-Frame)
0x50 (Response-Frame) usw.

Um die Komponenten des MAC-Datenfeldes einsichtig zu machen, müssen wir etwas ausholen.

1.3.1.2.1 Kommunikation zwischen den Layern

Genauso wie in der Intertask-Kommunikation, erfolgt die Kommunikation zwischen den Layern über **Mailboxen**. Die Anwendertasks befinden sich in der Privileg-Ebene 3 und die Layer in der Privileg-Ebene 0. Das Prinzip der Interlayer-Kommunikation zeigt Bild 21. Zwei Layer sind beteiligt: N+1 und N.

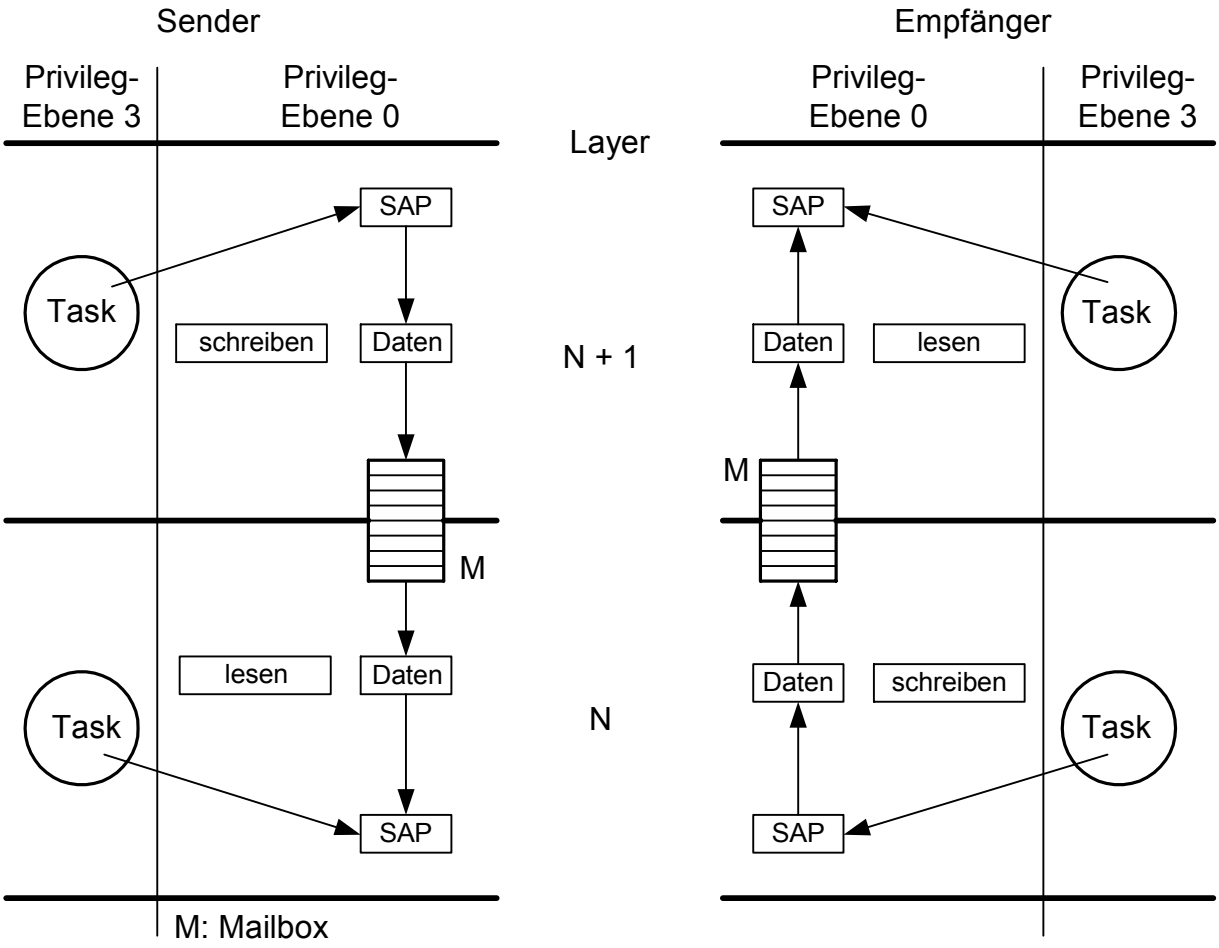


Bild 21: Prinzip der Interlayer-Kommunikation

Auf der Senderseite hat eine Task in der Privileg-Ebene 3 einige Daten, die sie an den Layer N+1 übergibt. Sie ruft zu diesem Zweck eine zugängliche Funktion in der Priveleg-Ebene 0 auf, die man allgemein als **Service Access Point** (SAP) bezeichnet. Solche SAPs stehen in jedem Layer des OSI-Referenzmodells (OSI = Open Systems Interconnection) zur Verfügung. Der Layer N+1 bearbeitet die Daten und gibt sie an den darunterliegenden Layer N weiter. Zu diesem Zweck schreibt er die Daten in eine Mailbox. Eine andere Task aktiviert einen SAP im Layer N. Dieser liest die Daten, bearbeitet sie und gibt sie an den nächsten Layer N-1 weiter. Dies geschieht solange, bis der unterste Layer (Physical Layer 1) erreicht ist.
Auf der Empfängerseite erfolgt die Kommunikation in umgekehrter Richtung. Eine Task aktiviert einen SAP im Layer N und dieser schreibt die Daten in die dortige Mailbox. Der Layer N+1 liest und bearbeitet sie, und die nächste Task kann sie danach ebenfalls über einen SAP empfangen.

Wir wollen uns als nächstes ansehen, wie zwei Seiten eines Netzwerkes miteinander kommunizieren, und was dabei mit den Daten auf ihrem vertikalen Weg durch die Layer auf den beiden Seiten des Netzwerkes geschieht. Eine pragmatische Illustration hierzu zeigt Bild 22. Es sind drei Layer beteiligt: N+1, N und N-1.

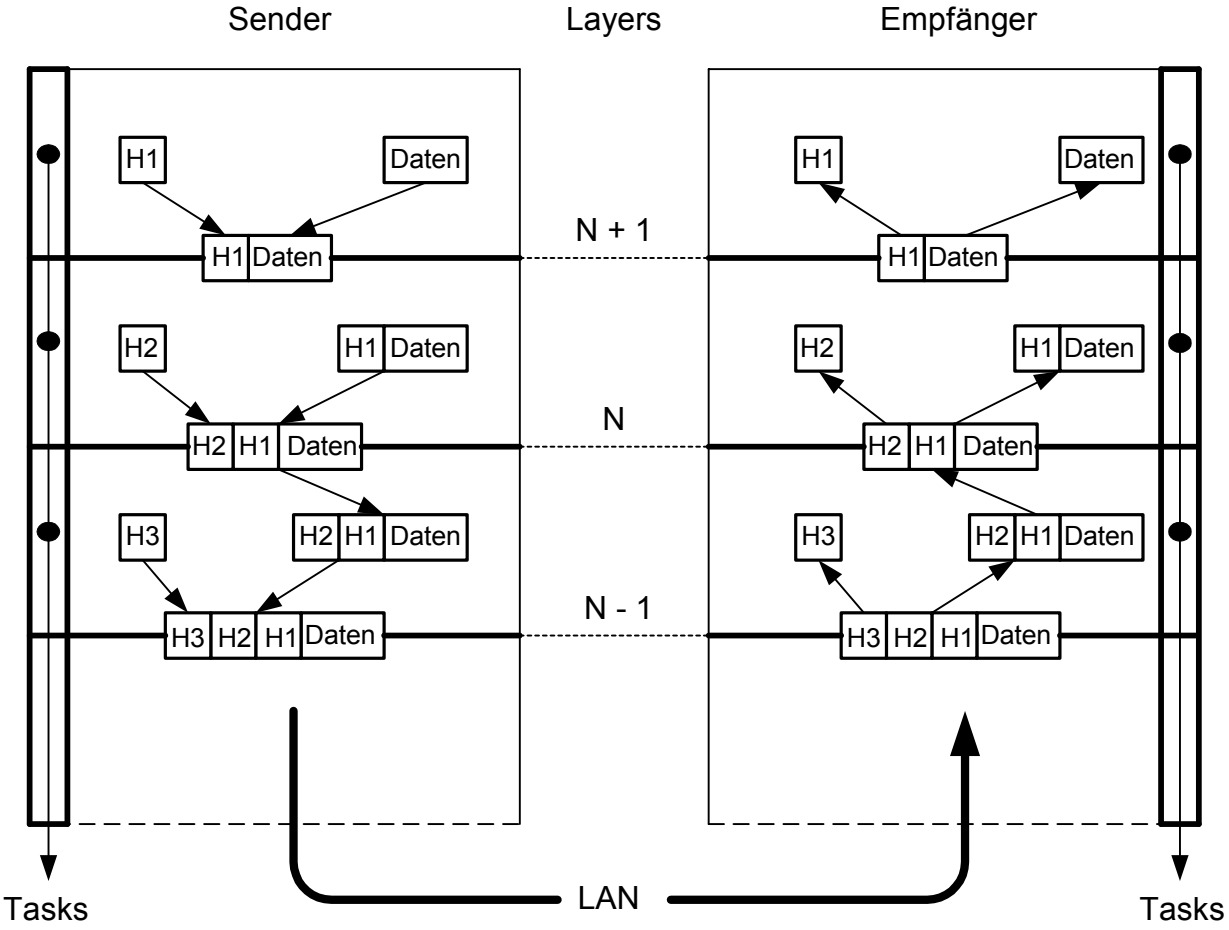


Bild 22: Kommunikation zwischen zwei Seiten in einem Netzwerk

In Anlehung an die Situation nach Bild 21, übergibt eine Sendetask an den Layer N+1 einige Daten. Dieser hängt vor die Daten einen sogenannten **Header** (H1), der aus einer Anzahl von Control-Informationen besteht. Manchmal werden die Daten als **SDU** (Service Data Unit) und der Header als **PCI** (Protocol Control Information) bezeichnet. Beide zusammen stellen die PDU dar (PDU = PCI+SDU).
Der Layer N+1 gibt nun die entstandene Einheit, also die PDU, über die Mailbox weiter an den Layer N. Dieser interpretiert die empfangene Einheit als SDU, und fügt seinerseits einen Header (H2) hinzu. Die neu entstandene PDU gibt der Layer N weiter an den Layer N-1, der sie wiederum als SDU interpretiert, und mit einem weiteren Header (H3) versieht. Schließlich kommen die originalen Daten, versehen mit drei Headern, beim Physical Layer an, und werden dort als Bitstrom zum Empfänger übertragen.
Der Layer N-1 auf der Empfängerseite entfernt aus der empfangenen PDU den Header H3 und gibt den Rest weiter an den Layer N. Dieser entfernt den Header H2 und gibt seinerseits den Rest weiter an den Layer N+1. Dort wird der Header H1 entfernt, und die originalen Daten treffen schließlich bei der Empfängertask ein.

Die PDUs, die in den individuellen Layern erzeugt werden, kommen uneingeschränkt bei den korrespondierenden Layern der Empfängerseite an. Für den Netzwerkprogrammierer sieht es so aus, wie wenn die Datenübertragung in **horizontaler** Richtung erfolgen würde. Da aber in Wahrheit die Datenübertragung in vertikaler Richtung erfolgt, spricht man von **virtuellen** Verbindungen zwischen den Layern auf der Sender- und Empfängerseite.
Kehren wir nun wieder zur MAC PDU zurück, die die Receive Machine im rxm_puffer[] hinterlegt hat. Unser unmittelbares Interesse gilt jetzt dem Aufbau des **MAC-Datenfeldes**. Seine Komponenten lassen sich unmittelbar aus den Kenntnissen über die Kommunikation zwischen den Layern erklären. Dazu sei folgendes angemerkt: Ein tokenbus-basiertes LAN kommt mit 4 Layern aus. Es benötigt

Layer 4: Die Transportschicht mit z.B. dem UDP-Protokoll (UDP = User Data Protocol)
Layer 2b: Die LLC-Schicht mit einem Sicherungsprotokoll (LLC = Logical Link Control)
Layer 2a: Die MAC-Schicht (MAC = Medium Access Control)
Layer 1: Die PHYS- bzw. Bitübertragungs-Schicht (PHYS = Physical)

Der Layer 3, die Vermittlungsschicht mit dem IP-Protokoll (IP = Internet Protocol), kommt nicht vor.
Aus dieser Topologie resultiert unmittelbar der Aufbau des MAC-Datenfeldes. Wie es sich gestaltet, zeigt das komplette Layout der MAC PDU im Bild 23.

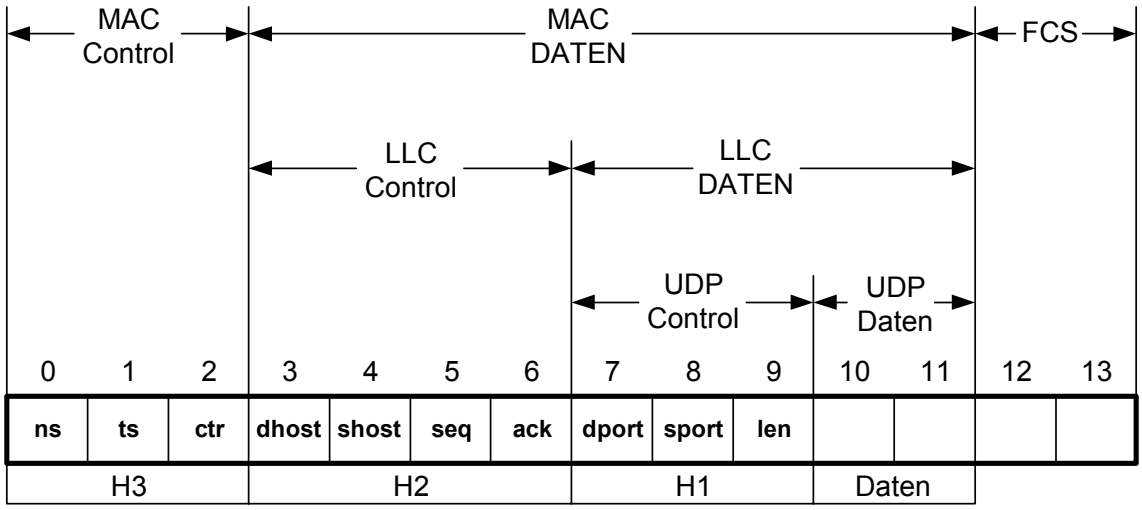


Bild 23: Komplettes Layout der MAC PDU

Das **MAC Control**-Feld entspricht dem Header **H3** und enthält die uns bereits bekannten Komponenten ns (next station), ts(this station) und ctr(control).
Das **MAC Daten**-Feld enthält Komponenten der nächsthöheren LLC-Schicht (Layer 2b).
Dies sind
- die LLC-Control-Informationen (Header H2) und
- die LLC-Daten.

Die LLC-Daten ihrerseits setzen sich aus den Komponenten der nächstfolgenden Transportschicht (Layer 4) zusammen. In Anlehnung an das dort benutzte UDP-Protokoll sind dies
- die UDP Control-Informationen (Header H1) und
- die UDP-Daten, die mit den originalen Daten der Anwendertask identisch sind.

Betrachten wir als erstes die LLC Control-Informationen. Dazu gehören die Komponenten

dhost: = **destination host** (1 Byte). Enthält die Adresse des Rechners, an den die Anwenderdaten geschickt werden sollen.

shost: = **source host** (1 Byte). Enthält die Adresse des Rechners, von dem die Anwenderdaten abgeschickt werden.

Anmerkung: Die Protokolle der MAC-Schicht und der LLC-Schicht verfolgen bei der Datenübertragung unterschiedliche Interessen. Daher kennzeichnet jede Schicht seine Station bzw. seinen Rechner mit einer individuellen Nummer. Das heißt ts (MAC) und shost(LLC) bzw. ns (MAC) und dhost (LLC) sind nicht identisch.

seq und ack: = **sequence** und **acknowledge** (je 1 Byte). Beide Felder benutzt das Sicherungsprotokoll der LLC-Schicht. Dieses Protokoll betrachten wir in einem späteren Kapitel.

Als nächstes seien die UDP Control-Informationen genannt. Dazu gehören die Komponenten

dport: = **destination port** (1 Byte). Enthält die Adresse einer Mailbox auf der Empfängerseite der Transportschicht. In diese Mailbox werden die UDP-Daten (originale Anwenderdaten) eingekettet und können dort Anschließend von der Empfangstask abgeholt werden.

sport: = **source port** (1 Byte). Enthält die Adresse einer Mailbox auf der Sendeseite der Transportschicht. In diese Mailbox werden die UDP-Daten (originale Anwenderdaten) der Sendetask eingekettet und gehen von dort auf die Reise durch Layer der Sendeseite, dem LAN und die Layer der Empfangsseite zur Empfangstask.

len: = length (1 Byte). Enthält die Anzahl der UDP-Daten.

Bleibt zum Schluß noch das UDP-Datenfeld. Es enthält die originalen Anwenderdaten und kann aus einem oder mehreren Oktets bestehen. Dies ist abhängig von der FIFO-Kapazität des UART. Bei einer Größe von 14 Oktets verbleiben noch 2 Oktets für die UDP-Daten.

1.3.1.3 Cyclic Redundancy Code (CRC)

Hat die Receive Machine die MAC PDU empfangen, ist nicht garantiert, daß alle Bits unbeschädigt angekommen sind. So ist die hardware-basierte Fehlererkennung durch den UART, speziell durch die Schwächen des Priortätsüberwachungsschemas, in seiner Leistungsfähigkeit eingeschränkt (siehe 1.3.1.1).
In der Praxis wird daher oft ein anderes oder auch zusätzliches Modell benutzt, nämlich der **Polynomcode**. Er ist auch als zyklischer Redundanzcode oder CRC (cyclic redundancy code) bekannt. Polynomcodes basieren auf der Idee, einen Bitstrom als Polynom mit der Basis x=2 und den Koeffizienten 0 und 1 zu behandeln. Zum Beispiel kann der Bitstrom

$$\begin{array}{cccccccc} 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{array}$$

als Polynom folgendermaßen dargestellt werden:

$$\mathbf{f(x) = 1x^7 + 1x^6 + 0x^5 + 0x^4 + 1x^3 + 0x^2 + 0x^1 + 1x^0}$$

oder kürzer

$$\mathbf{f(x) = x^7 + x^6 + x^3 + 1}$$

Wenn ein UART einen Frame (MAC PDU) überträgt, dann kann der korrespondierende Bitstrom als Polynom T(x) betrachtet werden, mit dem einige Berechnungen durchführbar sind. Von Interesse ist dabei die Frage, ob das Polynom T(x) durch ein anderes Polynom G(x) teilbar ist. Führt man eine solche Division durch, lautet das Ergebnis allgemein

$$\mathbf{T(x) : G(x) = I(x) + R(x)}$$

Man erhält das Polynom I(x) und das Restpolynmom R(x). Ist R(x) = 0, dann ist der Dividend durch den Divisor teilbar. Ist $R(x) \neq 0$, dann ist der Dividend durch den Divisor nicht teilbar. Was kann man mit dieser Erkenntnis anfangen? Sie ist der Schlüssel zu einer Methode, mit der sich Übertragungsfehler erkennen lassen. Betrachten wir dazu ein Beispiel aus dem Zehnersystem. Siehe Bild 24.

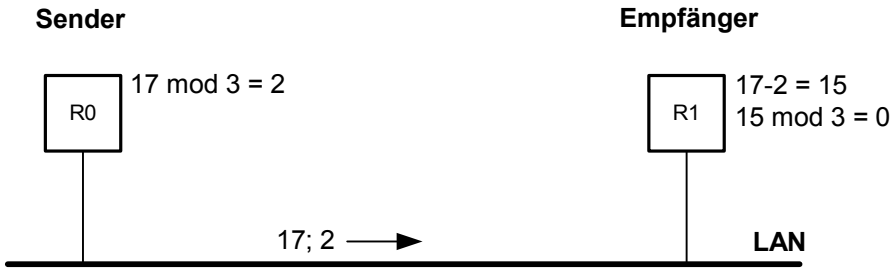


Bild 24: Methode zur Erkennung von Übertragungsfehlern

Der Sender R0 hat die Zahl 17, die beim Empfänger R1 fehlerfrei ankommen soll. Sowohl der Sender als auch der Empfänger führen eine modulo Division durch, und einigen sich zu diesem Zweck auf einen gemeinsamen Divisor, z.B. 3.

Der Sender berechnet 17:3 = 5 Rest 2, und schickt den Dividenden 17 und den Rest 2 über das LAN zum Empfänger. Der Empfänger subtrahiert vom empfangenen Dividenden den Rest, also 17-2 = 15, und führt anschließend mit dem Ergebnis die Division 15:3 = 5 Rest 0 durch. Der **Rest 0** ist ein Indiz dafür, daß kein Übertragungsfehler aufgetreten ist.
Um diese Methode für die Praxis nutzen zu können, müssen wir sie auf Polynomberechnungen im binären Zahlensystem anwenden. Dazu benötigen wir die Regeln der modulo 2 Rechnung. Eine allgemeine Betrachtung über die modulo m Rechnung ist daher von Vorteil.

1.3.1.3.1 Die modulo m Rechnung

Es sei M = {-14,-4,6,16,26} eine Menge von Zahlen im Zehnersystem. Zu jeder Zahl x suchen wir die „Restzahl" r, und führen zu diesem Zweck die modulo m Rechnung aus.

Das Verfahren lautet: Subtrahiere **x-m** oder addiere **x+m** solange, bis eine Restzahl r entsteht, die in der sogenannten Restklasse **0...m-1** enthalten ist.

modulo 10 Rechnung:

x modulo 10	= r
-14 modulo 10 = -14+10 = -4; -4+10	= 6
-4 modulo 10 = -4+10	= 6
6 modulo 10	= 6
16 modulo 10 = 16-10	= 6
26 modulo 10 = 26-10 = 16; 16-10	= 6

Was sagen uns die Ergebnisse? In der modulo m Rechnung wird nicht zwischen der Zahl x und der Zahl r unterschieden. So gilt für das angegebene Beispiel

6 = {...,-14,-4,6,16,26,...}

daß alle rechts stehenden Zahlen durch die Zahl 6 repräsentiert werden. Dies kann man allgemein folgendermaßen ausdrücken:

r = r+k•m; k =...,-2,-1,0,+1,+2,...

Verifizierung:

6 = 6-2•10 = -14 (k = -2)
6 = 6-1•10 = -4 (k = -1)
6 = 6+0•10 = 6 (k = 0)
6 = 6+1•10 = 16 (k =+1)
6 = 6+2•10 = 26 (k =+2)

Die Schlußfolgerung lautet: In der modulo m Rechnung gibt es genau m unterscheidbare Zahlen. Es sind dies die Elemente 0..m-1 eines Körpers, den man auch als Galois-Feld GF(m) bezeichnet. Im GF(10) sind es die Zahlen **0,1,.....9**. Im GF(2) sind es die Zahlen **0,1**.

1.3.1.3.2 Die Arithmetik im Galois-Feld GF(2)

Wir führen eine modulo2 Addition und eine modulo2 Subtraktion durch, und überprüfen bei beiden Rechnungsarten, ob die Ergebnisse im GF(2) liegen.

Addition:

0+0 = 0; 0 modulo2 = 0
0+1 = 1; 1 modulo2 = 1
1+0 = 1; 1 modulo2 = 1
1+1 = 2; 2 modulo2 = 0

Subtraktion:

0-0 = 0; 0 modulo2 = 0
0-1 = -1; -1 modulo2 = 1
1-0 = 1; 1 modulo2 = 1
1-1 = 0; 0 modulo2 = 0

Bei der modulo2 Addition werden die Ergebnisse der Operationen 0+0 und 1+1 durch 0 repräsentiert. Die Ergebnisse der Operation 0+1 und 1+0 durch 1.
Bei der modulo2 Subtraktion werden die Ergebnisse der Operationen 0-0 und 1-1 durch 0 repräsentiert. Die Ergebnisse der Operationen 0-1 und 1-0 durch 1.
Also liegt bei beiden Rechnungsarten ein Galois-Feld GF(2) mit den Elementen M={0,1} vor. Desweiteren zeigen die Ergebnisse, daß es zwischen der modulo2 Addition und der modulo2 Subtraktion keinen Unterschied gibt. Die Operationen + und - lassen sich somit durch EXOR-Verknüpfungen realisieren.

Beispiele:

modulo2 Addition	modulo2 Subtraktion
1011 0100	0101 0000
+1101 0000 (EXOR)	-1000 0001 (EXOR)
0110 0100	1101 0001

Kehren wir zurück zur Polynomberechnung. Wir benutzen die Regeln der modulo2 Arithmetik und führen damit die Division T(x) : G(x) = I(x) + R(x) durch.

Beispiel 1:

Es sei **$T(x) = 1x^4+1x^3+0x^2+1x^1+1x^0$** und **$G(x) = 1x^2+1x^1+1x^0$**. Gesucht wird das Restpolynom **R(x)**.

Division T(x) durch G(x):

T(x) : G(x) = I(x) + R(x)

$1x^4+1x^3+0x^2+1x^1+1x^0 : 1x^2+1x^1+1x^0 = 1x^2+0x^1+1x^0 + 0$
$\underline{1x^4+1x^3+1x^2}$ ↓ ↓
$\mathbf{1x^2}+1x^1+1x^0$
$\underline{1x^2+1x^1+1x^0}$
Rest: 0

Im Beispiel 1 ist wegen R(x)=0 der Dividend T(x) durch den Divisor G(x) teilbar. Außerdem ist im Sinne der modulo m Rechnung zwischen dem „Polynom 0“ und dem Polynom $T(x)=1x^4+1x^3+0x^2+1x^1+1x^0$ nicht zu unterscheiden.

Beispiel 2:

Es sei **$T(x) = 1x^5+0x^4+1x^3+0x^2+0x^1+1x^0$** und **$G(x) = 1x^2+1x^1+1x^0$**. Gesucht wird auch hier das Restpolynom **R(x)**.

Division T(x) durch G(x):

T(x)	:	G(x)	=	I(x)	+	R(x)

$1x^5+0x^4+1x^3+0x^2+0x^1+1x^0 : 1x^2+1x^1+1x^0 = 1x^3+1x^2+1x^1 + x+1$

$\underline{1x^5+1x^4+1x^3}$ ↓ ↓ ↓

$\mathbf{1x^4}+\mathbf{0x^3}+0x^2+0x^1+1x^0$

$\underline{1x^4+1x^3+1x^2}$ ↓ ↓

$\mathbf{1x^3}+\mathbf{1x^2}+0x^1+1x^0$

$\underline{1x^3+1x^2+1x^1}$ ↓

Rest: $\mathbf{1x^1}+1x^0$ bzw. x+1

Im Beispiel 2 ist wegen R(x)=x+1 der Dividend T(x) durch den Divisor G(x) **nicht** teilbar. Auch hier gilt natürlich, daß es zwischen T(x) und R(x) keine Unterscheidung gibt.

1.3.1.3.3 Fehlererkennung nach der Polynomcodemethode

Die Polynomcodemethode benutzt die Polynomdivision, wie wir sie anhand von zwei Beispielen kennengelernt haben. In Anlehnung an die Methode nach Bild 24, einigen sich der Sender und der Empfänger auf einen gemeinsamen Divisor, dem sogenannten **Generatorpolynom** G(x). Dabei müssen in G(x) das höchstwertigste und das niederwertigste Bit gleich 1 sein. Die Grundidee ist, auf der Senderseite an den originalen Bitstrom, der dem Polynom T(x) entspricht, eine Prüfsumme so anzuhängen, daß sich auf der Empfängerseite das resultierende Polynom durch G(x) teilen läßt:

Der detailierte Algorithmus lautet wie folgt:

1, Sender:

- Wenn k der Grad von G(x) ist, dann füge k Nullbits an T(x) an, so daß das Polynom $T(x)x^k$ entsteht. Ist n die Anzahl der Bits in T(x), dann enthält das resultierende Polynom n+k Bits.

  Beispiel: Sei $T(x) = 1x^3+0x^2+1x^1+0x^0 = x^3+x$ und
  $G(x) = 1x^3+0x^2+1x^1+1x^0 = x^3+x+1$

  Der Grad k von G(x) ist 3. Somit wird

  $$T(x)x^k = (1x^3+0x^2+1x^1+0x^0)x^3 = \underbrace{1x^6+0x^5+1x^4+0x^3}_{n=4} + \underbrace{0x^2+0x^1+0x^0}_{k=3}$$

- Führe die modulo2 Division $T(x)x^k : G(x)$ aus.

  Beispiel Fortsetzung:

  $$T(x)x^k \quad : \quad G(x) \quad = \quad I(x) \quad + \quad R(x)$$

  $$1x^6+0x^5+1x^4+0x^3+0x^2+0x^1+0x^0 : 1x^3+0x^2+1x^1+1x^0 = 1x^3+0x^2+0x^1+1x^0 \quad + \quad x+1$$

  $$\underline{1x^6+0x^5+1x^4+1x^3} \quad \downarrow \quad \downarrow \quad \downarrow$$

  $$\mathbf{1x^3}+0x^2+0x^1+0x^0$$

  $$\underline{1x^3+0x^2+1x^1+1x^0}$$

  Rest: $1x^1+1x^0$ bzw. **x+1**

  Der entstandene Rest x+1 ist die gesuchte Prüfsumme, die auch als **frame check sequence** (fcs) bekannt ist.

- Ziehe den Rest, also die fcs, durch modulo2 Subtraktion von $T(x)x^k$ ab. Das Ergebnis-Polynom, wir nennen es C(x), ist ein Polynomcode, der als **cyclic redundancy code** (crc) bezeichnet wird. Warum crc ein zyklischer Code ist, werden wir etwas später klären. **Hinweis**: Weil in der modulo2 Arithmetik zwischen Subtraktion und Addition nicht unterschieden wird, erscheint in der folgenden Rechnung +R(x).

  Beispiel Fortsetzung:

  $$T(x)x^k \quad + \quad R(x) \quad = \quad C(x)$$

  $$1x^6+0x^5+1x^4+0x^3+0x^2+0x^1+0x^0 \quad + \underbrace{1x^1+1x^0}_{\text{fcs}} = \underbrace{1x^6+0x^5+1x^4+0x^3+0x^2+\mathbf{1x^1}+\mathbf{1x^0}}_{\text{crc}}$$

Das vom Sender berechnete Polynom C(x) schickt nun sein UART als Bitstrom durch das LAN zum Empfänger.

2, Empfänger:

- Führe die modulo2 Division C(x) : G(x) aus.

  Beispiel Fortsetzung:

  $$C(x) \quad : \quad G(x) \quad = \quad I(x) \quad + \quad R(x)$$

  $$1x^6+0x^5+1x^4+0x^3+0x^2+1x^1+1x^0 \quad : \quad 1x^3+0x^2+1x^1+1x^0 \quad = \quad 1x^3+0x^2+0x^1+1x^0 \quad + \quad 0$$

  $$\underline{1x^6+0x^5+1x^4+1x^3} \quad \downarrow \quad \downarrow \quad \downarrow$$

  $$\mathbf{1x^3}+0x^2+1x^1+1x^0$$

  $$\underline{1x^3+0x^2+1x^1+1x^0}$$

  Rest: 0

Solange C(x) ein zulässiges Codewortpolynom ist, ist eine restfreie Division durch G(x) möglich, dann ist R(x)=0. Ein fehlerhaftes Codewortpolynom erkennt man daran, daß C(x) modulo G(x) ≠ 0 ist.
In unserem Beispiel ist R(x)=0 bzw. die fcs=0, und damit ist der Bitstrom unverstümmelt beim Empfänger angekommen. Ist das wirklich so? Kann beim Empfänger auch ein fehlerbehafteter Polynomcode ankommen, der trotzdem zu R(x)=0 führt?
Wenn C(x) auf dem Weg zum Empfänger beschädigt wird, dann addiert sich zu C(x) das Fehlerpolynom F(x). Mit anderen Worten, beim Empfänger kommt C(x)+F(x) an. Ist zufälligerweise F(x)=G(x), dann berechnet der Empfänger (modulo2)

$$(C(x)+F(x)) : G(x) = \frac{C(x)}{G(x)} + \frac{F(x)}{G(x)} = \frac{C(x)}{G(x)} + \frac{G(x)}{G(x)} = \mathbf{0} + \mathbf{0}$$

Die modulo2 Division $\frac{C(x)}{G(x)}$ ist immer 0, und die modulo2 Division $\frac{F(x)}{G(x)} = \frac{G(x)}{G(x)}$ ebenfalls.
Das heißt, Fehler die dem Polynom G(x) entsprechen, können nicht entdeckt werden, alle anderen werden dagegen erkannt. Doch wie oft kommt dies vor? Trotz geringer Wahrscheinlichkeit ist es gut, zusätzliche Fehlerüberwachungssysteme zu benutzen (z.B. die Paritätsüberprüfung des UART).

1.3.1.3.4 Zyklische Eigenschaften

Wir rotieren das originale Codewortpolynom C(x) und 1 Bitstelle links- oder rechtsherum, und bekommen das Polynom $C^{(1)}(x)$.

Beispiel: 1te Linksrotation

$$C(x) \rightarrow C^{(1)}(x)$$

$$1x^6+0x^5+1x^4+0x^3+0x^2+1x^1+1x^0 \rightarrow \mathbf{0x^6+1x^5+0x^4+0x^3+1x^2+1x^1+1x^0}$$

Wir führen die modulo2 Division $C^{(1)}(x) : G(x) = R(x)$ aus und bekommen

Beispiel: Fortsetzung

$$C^{(1)}(x) : G(x) = I(x) + R(x)$$

$$1x^5+0x^4+0x^3+1x^2+1x^1+1x^0 : 1x^3+0x^2+1x^1+1x^0 = 1x^2+0x^1+1x^0 + 0$$

$$\underline{1x^5+0x^4+1x^3+1x^2} \quad \downarrow \quad \downarrow$$
$$\mathbf{1x^3+0x^2}+1x^1+1x^0$$
$$\underline{1x^3+0x^2+1x^1+1x^0}$$
$$\text{Rest:} \quad 0$$

$C^{(1)}(x)$ ist offensichtlich ein gültiges Codewortpolynom, denn $C^{(1)}(x)$ modulo G(x) = 0.
Ist k der Grad von C(x), dann entstehen weitere gültige Codewortpolynome durch zyklische Rotation von C(x) um 2,3,..k Stellen. Nach der (k+1)ten Rotation entsteht dann wieder das Originalpolynom C(x). Es gibt also genau k+1 redundante Codewortpolynome.

Dies ist wohl der Grund für die Bezeichnung cyclic redundancy code. In unserem Beispiel ist der Grad k von C(x) = 6. Daher gibt es k+1=7 gültige Codewortpolynome. Hier eine Zusammenfassung:

1		$\mathbf{C(x) = 1x^6+0x^5+1x^4+0x^3+0x^2+1x^1+1x^0}$
2	1te Linksrotation:	$C^{(1)}(x) = 0x^6+1x^5+0x^4+0x^3+1x^2+1x^1+1x^0$
3	2te Linksrotation:	$C^{(2)}(x) = 1x^6+0x^5+0x^4+1x^3+1x^2+1x^1+0x^0$
4	3te Linksrotation:	$C^{(3)}(x) = 0x^6+0x^5+1x^4+1x^3+1x^2+0x^1+1x^0$
5	4te Linksrotation:	$C^{(4)}(x) = 0x^6+1x^5+1x^4+1x^3+0x^2+1x^1+0x^0$
6	5te Linksrotation:	$C^{(5)}(x) = 1x^6+1x^5+1x^4+0x^3+1x^2+0x^1+0x^0$
7	6te Linksrotation:	$C^{(6)}(x) = 1x^6+1x^5+0x^4+1x^3+0x^2+0x^1+1x^0$

7te Linksrotation: $C^{(7)}(x) = \mathbf{C(x) = 1x^6+0x^5+1x^4+0x^3+0x^2+1x^1+1x^0}$

1.3.1.3.5 Modulo2 Division und Hardware

Wir wollen jetzt als nächsten Schritt die uns bereits bekannte, vom Empfänger durchgeführte modulo2 Division, durch eine Hardware-Logik realisieren. Zur Erinnerung:

C(x) : G(x) = I(x) + R(x)

$1x^6+0x^5+1x^4+0x^3+0x^2+1x^1+1x^0 : 1x^3+0x^2+1x^1+1x^0 = 1x^3+0x^2+0x^1+1x^0 + 0$

$\underline{1x^6+0x^5+1x^4+1x^3}$ ↓ ↓ ↓

$\mathbf{1x^3}+0x^2+1x^1+1x^0$

$\underline{1x^3+0x^2+1x^1+1x^0}$

Rest: 0

Im Bild 25 ist diese Logik zu sehen.

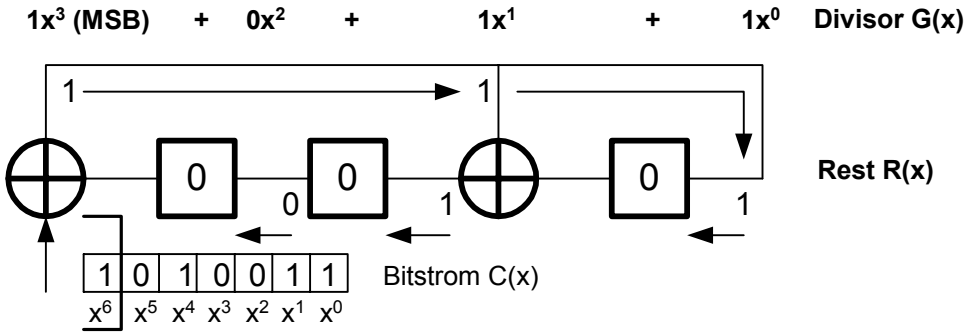


Bild 25: Modulo2 Divisions-Hardware

Sie besteht aus einer Anzahl von FlipFlops (□) und XORs (⊕). Die FlipFlops sind zu einem Schieberegister konfiguriert, das die temporären Restwerte während des Divisonsvorgangs aufnimmt, und am Ende der Division die fcs enthält. Anfänglich sind die FlipFlops gelöscht, das Schieberegister also leer, und damit die fcs=0. Die Anzahl der FlipFlops ist abhängig von der Anzahl der + Zeichen im Divisor G(x). An jeder Position eines +Zeichens erscheint im Schieberegister ein FlipFlop. Die Anzahl der XORs ist abhängig von den Koeffizienten der Polynomglieder im Divisor G(x). Ist der Koeffizient = 1, reiht sich an der betreffenden Stelle in die FlipFlop-Kette ein XOR ein. Dies ist an den Stellen $1x^3$, $1x^1$ und $1x^0$ der Fall. Weil an der Stelle $1x^0$ keine Verknüpfung stattfindet, ist es dort weggelassen.

Das XOR an der Stelle $1x^3$ hat zwei Eingänge. Der eine Eingang bekommt das höchstwertige Bit des Schieberegisters, und der andere Eingang die seriell einlaufenden Bits des Dividenden C(x).

1. Schritt: Die modulo2 Divisions-Hardware beginnt ihre Arbeit an der Divisorstelle $1x^3$ mit der XOR-Verknüpfung des höchstwertigen Bits in C(x) und dem höchstwertigen Bit des Schieberegisters: **1+0=1**, bzw. MSB=1. Über die Verbindungsleitung gelangt das MSB zu den Divisorstellen $1x^1$ und $1x^0$. Damit steht es an dem einen Eingang des zweiten XORs und am Eingang des niederwertigsten FlipFlops an. Das zweite XOR verknüpft das MSB mit dem Ausgang des niederwertigsten FlipFlops und liefert 1+0=1. Danach stehen an den Eingängen der drei FlipFlops neue Informationen an, nämlich 011. Sie entsprechen dem temporären Rest $R(x) = 0x^2+1x^1+1x^0$. Siehe Bild 25. Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 1te Bearbeitungsschritt ist beendet. Siehe Bild 25a.

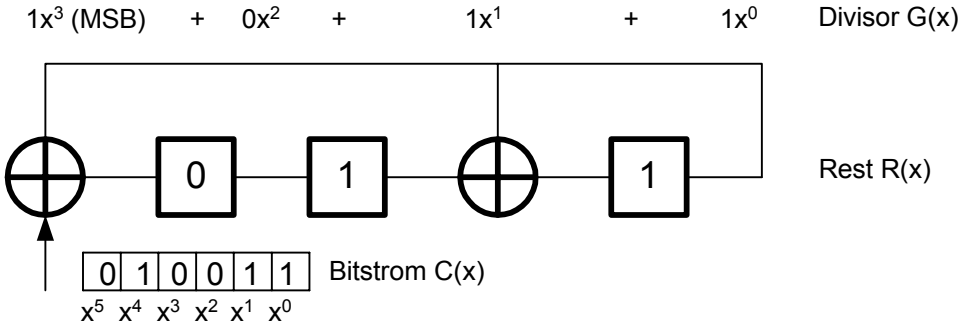


Bild 25a: Situation **nach** der Bearbeitung von $1x^6$ des Bitstroms C(x)

2. Schritt: Es folgt die Bearbeitung von Bit $0x^5$ des seriellen Bitstroms C(x):

- Die XOR-Verknüpfung an der Divisorstelle $1x^3$ liefert MSB=0: **0+0=0**,
- die XOR-Verknüpfung an der Divisorstelle $1x^1$ ergibt: **0+1=1**,
- der temporäre Rest lautet $R(x)=\mathbf{1x^2+1x^1+0x^0}$. Siehe Bild 25b.

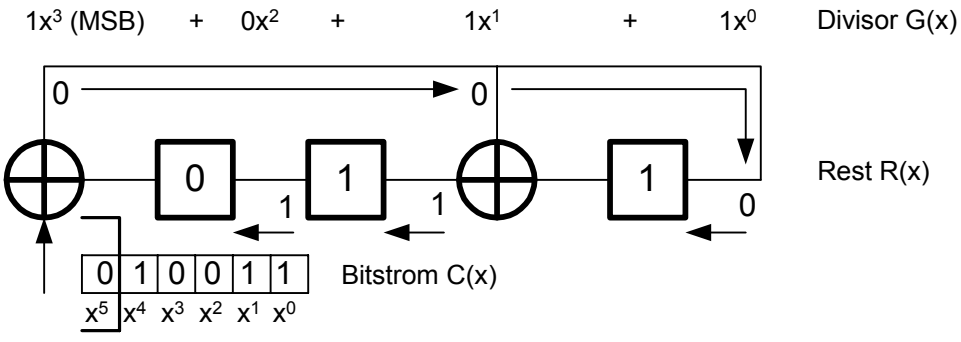


Bild 25b: Bearbeitung von Bit $0x^5$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 2te Bearbeitungsschritt ist beendet. Siehe Bild 25c.

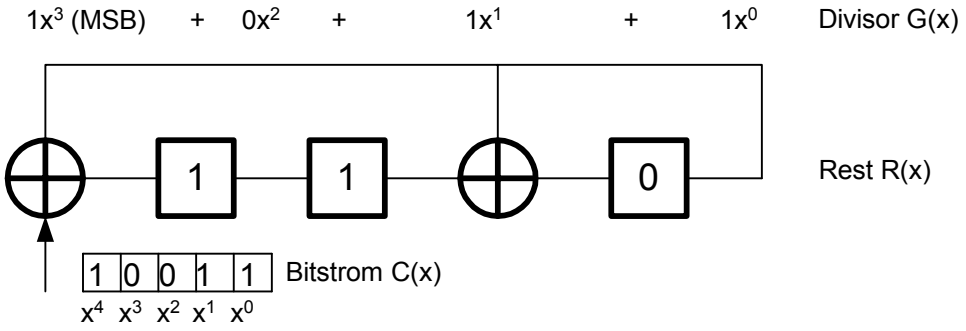


Bild 25c: Situation **nach** der Bearbeitung von $0x^5$ des Bitstroms C(x)

3. Schritt: Es folgt die Bearbeitung von Bit $1x^4$ des seriellen Bitstroms C(x):

- Die XOR-Verknüpfung an der Divisorstelle $1x^3$ liefert MSB=0: **1+1=0**,
- die XOR-Verknüpfung an der Divisorstelle $1x^1$ ergibt: **0+0=0**,
- der temporäre Rest lautet R(x)=$\mathbf{1x^2+0x^1+0x^0}$. Siehe Bild 25d.

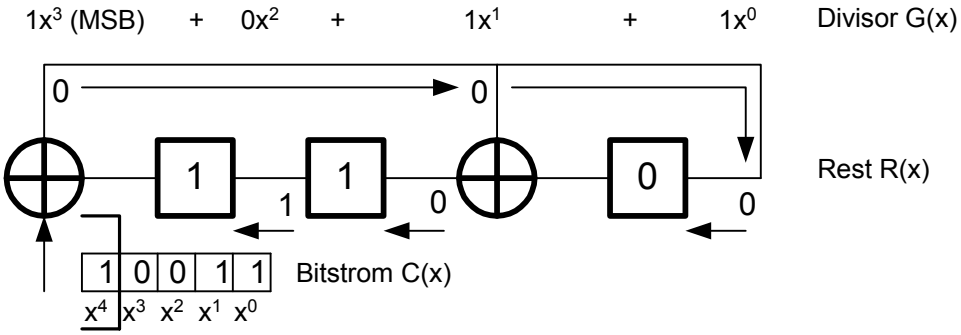


Bild 25d: Bearbeitung von Bit $1x^4$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 3te Bearbeitungsschritt ist beendet. Siehe Bild 25e.

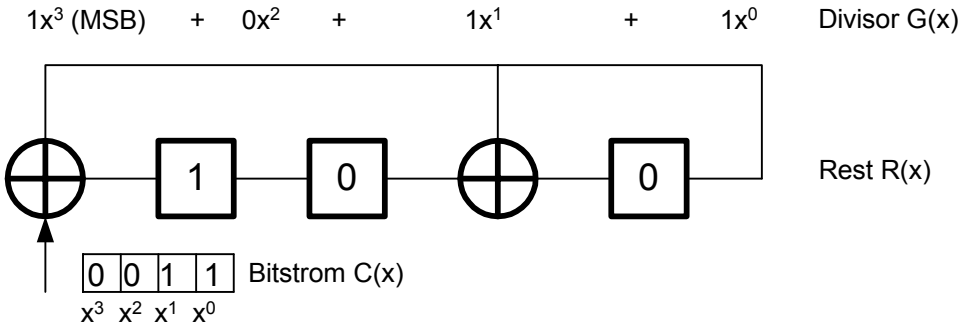


Bild 25e: Situation **nach** der Bearbeitung von $1x^4$ des Bitstroms C(x)

4. Schritt: Es folgt die Bearbeitung von Bit $0x^3$ des seriellen Bitstroms C(x):

- Die XOR-Verknüpfung an der Divisorstelle $1x^3$ liefert MSB=0: **0+1=1**,
- die XOR-Verknüpfung an der Divisorstelle $1x^1$ ergibt: **1+0=1**,
- der temporäre Rest lautet R(x)=$\mathbf{0x^2+1x^1+1x^0}$. Siehe Bild 25f.

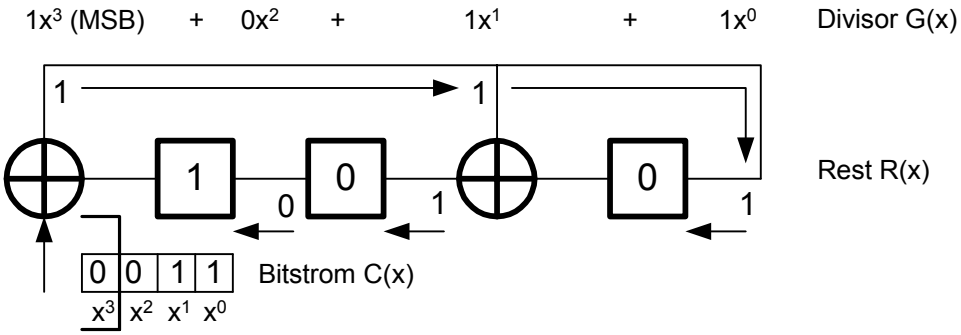


Bild 25f: Bearbeitung von Bit $0x^3$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 3te Bearbeitungsschritt ist beendet. Siehe Bild 25g.

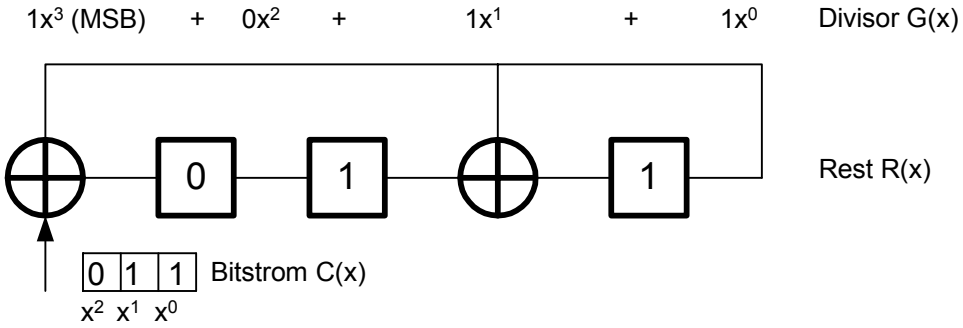


Bild 25g: Situation **nach** der Bearbeitung von $0x^3$ des Bitstroms C(x)

5. Schritt: Es folgt die Bearbeitung von Bit 0x2 des seriellen Bitstroms C(x):

- Die XOR-Verknüpfung an der Divisorstelle $1x^3$ liefert MSB=0: **0+0=0**,
- die XOR-Verknüpfung an der Divisorstelle $1x^1$ ergibt: **0+1=1**,
- der temporäre Rest lautet R(x)=$\mathbf{1x^2+1x^1+0x^0}$. Siehe Bild 25h.

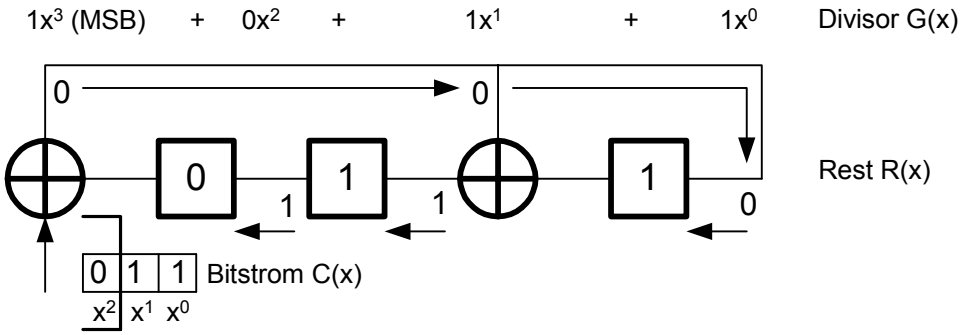


Bild 25h: Bearbeitung von Bit $0x^2$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 5te Bearbeitungsschritt ist beendet. Siehe Bild 25i.

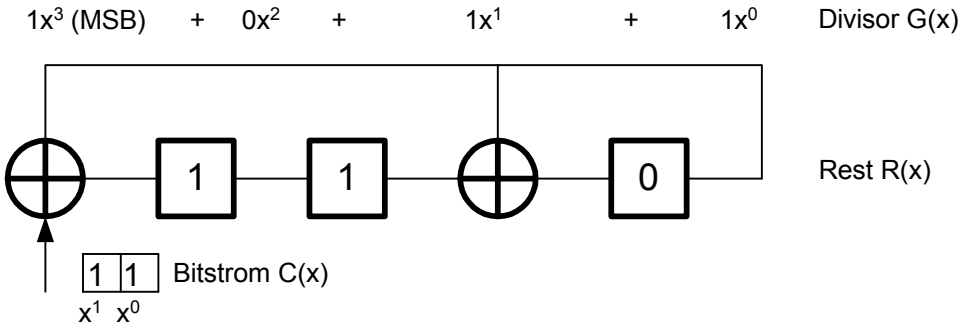


Bild 25i: Situation **nach** der Bearbeitung von $0x^2$ des Bitstroms C(x)

6. Schritt: Es folgt die Bearbeitung von Bit $1x^1$ des seriellen Bitstroms C(x):

- Die XOR-Verknüpfung an der Divisorstelle $1x^3$ liefert MSB=0: **1+1=0**,
- die XOR-Verknüpfung an der Divisorstelle $1x^1$ ergibt: **0+0=0**,
- der temporäre Rest lautet R(x)=$\mathbf{1x^2+0x^1+0x^0}$. Siehe Bild 25j.

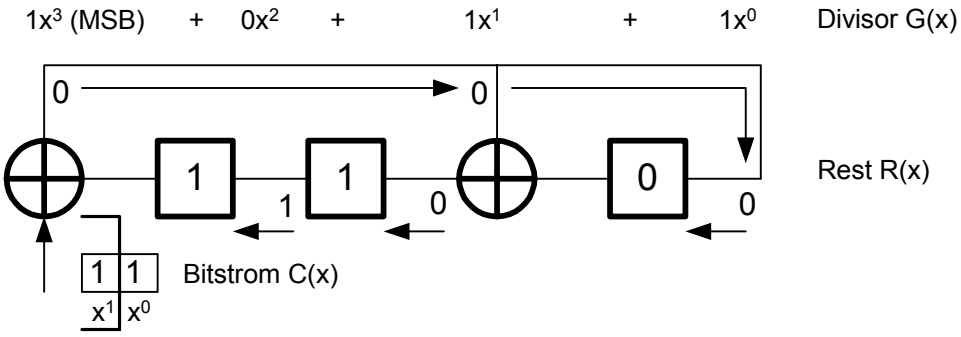


Bild 25j: Bearbeitung von Bit $1x^1$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 6te Bearbeitungsschritt ist beendet. Siehe Bild 25k.

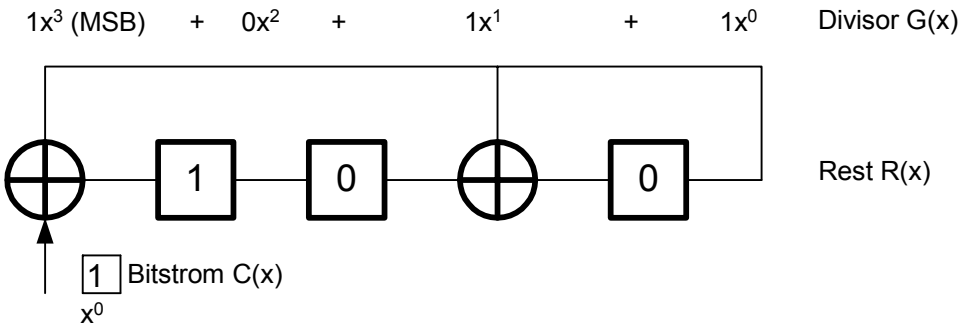


Bild 25k: Situation nach der Bearbeitung von $1x^1$ des Bitstroms C(x)

7. Schritt: Es folgt die Bearbeitung von Bit $1x^0$ des seriellen Bitstroms C(x):

- Die XOR-Verknüpfung an der Divisorstelle $1x^3$ liefert MSB=0: **1+1=0**,
- die XOR-Verknüpfung an der Divisorstelle $1x^1$ ergibt: **0+0=0**,
- der temporäre Rest lautet R(x)=$\mathbf{0x^2+0x^1+0x^0}$. Siehe Bild 25l.

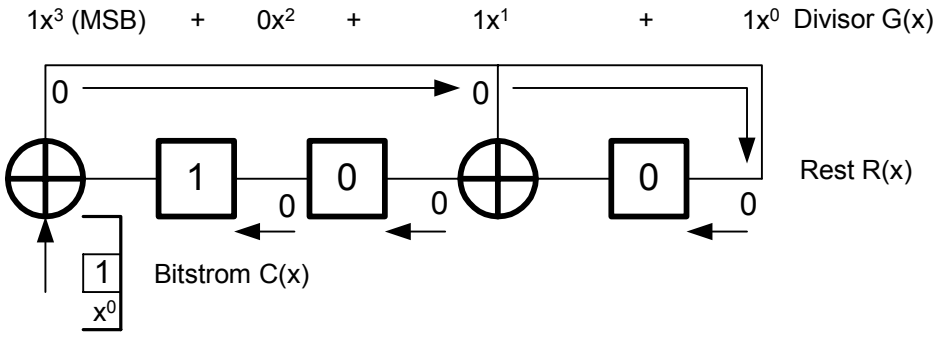


Bild 25l: Bearbeitung von Bit $1x^0$ des Bitstroms C(x)

Durch Linksschieben um 1 Bitposition wird das temporäre R(x) zur weiteren Verarbeitung in die FlipFlops übertragen. Der 7te Bearbeitungsschritt ist beendet. Gleichzeitig ist auch das Bitreservoir im Codewortpolynom C(x) ausgeschöpft.
Die Division C(x) modulo G(x) = 0 ist damit abgeschlossen. Die Hardware hat das erwartete Ergebnis R(x)=0 bzw. fcs=0 geliefert. Siehe Bild 25m.

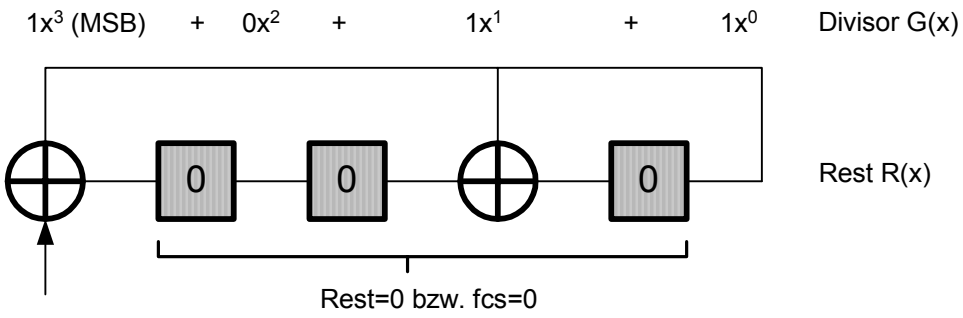


Bild 25m: Das Ergebnis der Hardware-Division C(x) modulo G(x)=0

1.3.1.3.6 Modulo2 Division und Software

Wir wollen ein C-Programm entwickeln, das die Mechanismen der modulo2 Divisions-Hardware simuliert.

1, Vorbereitung:

Bilde aus dem Divisor $G(x)=1x^3+0x^2+1x^1+1x^0$ die Maske $m(x)=\mathbf{0}x^3+0x^2+1x^1+1x^0$.
Mit anderen Worten, ersetze den Koeffizienten 1 im höchstwertigen Polynomglied des Divisors durch 0.

2, Algorithmus:

Ermittle das MSB.

a, Wenn **MSB=1** berechne:

- Augenblicklichen Inhalt des Schieberegisters R(x) linksschieben;
  $R(x)<< \rightarrow R'(x)$
- addiere (modulo2) zu $R'(x)$ die Maske m(x);
  $R'(x) + m(x)$
- das Ergebnis ist das neue R(x).

Beispiel: $R(x) = 0x^2+0x^1+0x^0$;augenblickliches R(x), vergleiche mit **Bild 25**
$R'(x) = 0x^2+0x^1+0x^0$;R(x)<<

$$R'(x) + m(x) = \begin{array}{r} 0x^2+0x^1+0x^0 \\ +\ \underline{0x^2+1x^1+1x^0} \\ \mathbf{0x^2+1x^2+1x^2} \end{array}$$;neues R(x)

Das neue $R(x)= 0x^2+1x^2+1x^2$ ist mit dem Ergebnis der modulo2 Hardware-Division identisch. Vergleiche mit **Bild 25a**.

b, Wenn **MSB=0**

- schiebe den augenblicklichen Inhalt des Schieberegisters um 1 Bitposition nach links.

Beispiel: $R(x) = 0x^2+1x^1+1x^0$;augenblickliches R(x), vergleiche mit **Bild 25b**
$R(x)<< \rightarrow 1x^2+1x^1+0x^0$;neues R(x)

Das neue $R(x)= 1x^2+1x^2+0x^2$ ist mit dem Ergebnis der modulo2 Hardware-Division identisch. Vergleiche mit **Bild 25c**.

1.3.1.3.7 fcs-Berechnung für 1 Oktet

Das folgende C-Programm (siehe Bild 26) wird sowohl von der Transmit Machine als auch von der Receive Machine des MAC-Layers benutzt. Es realisiert den oben beschriebenen Algorithmus mit dem realen Generator-Polynom CRC-16, und liefert eine 16-Bit frame check sequence für einen Bitstrom, der aus 8 Bits (1 Oktet) besteht.
Das CRC-16 Generator-Polynom hat den Grad k=16, besteht also aus 17 Bits und hat folgende Form:

CRC-16:
$\mathbf{1x^{16}+0x^{15}+0x^{14}+0x^{13}+1x^{12}+0x^{11}+0x^{10}+0x^{9}+0x^{8}+0x^{7}+0x^{6}+1x^{5}+0x^{4}+0x^{3}+0x^{2}+0x^{1}+1x^{0}}$

Daraus ergibt sich die Maske (Zur Erinnerung: Den Koeffizienten an der höchstwertigen Stelle x^{16} 0 setzen):

mask:
$\mathbf{0x^{16}+0x^{15}+0x^{14}+0x^{13}+1x^{12}+0x^{11}+0x^{10}+0x^{9}+0x^{8}+0x^{7}+0x^{6}+1x^{5}+0x^{4}+0x^{3}+0x^{2}+0x^{1}+1x^{0}}$

Hinweis: Die IEEE 802.4 empfiehlt ein Generator-Polynom mit dem Grad k=32.

Das C-Programm enthält die Funktion **crc**, die mit drei Parametern aufgerufen wird:

- unsigned short int fcs ;den temporären Rest, der den anfänglichen Wert 0 hat und ;in einem 16 Bit-Datentyp enthalten ist,
- unsigned short int c ;den Bitstrom (1 Oktet), der sich im Low-Byte eines ;16 Bit-Datentyps aufhält, und
- unsigned short int mask ;der Maske, die ebenfalls in einem 16 Bit-Datentyp ;aufbewahrt ist.

```
unsigned short int crc(unsigned short int fcs,unsigned short int c,unsigned short int mask)
{
 unsigned char i;

 c<<=8;
 for(i=0;i<8;i++)
     {
      if((fcs ^ c) & 0x8000) fcs = (fcs<<1)^mask;
      else fcs<<=1;
      c<<=1;
     }
 return fcs;
}
```

Bild 26: fcs-Berechnung für 1 Oktet

Die Funktion beginnt ihre Arbeit, indem sie als erstes die Variable c um 8 Bitpositionen nach links schiebt, um damit den Bitstrom (1 Oktet), der sich ja im Low-Byte der Variablen c aufhält, an die höchstwertige Stelle x^{16} des Generator-Polynoms zu bringen: **c<<=8**;

Bei einer anfänglichen fcs=0 entsteht dann folgende Situation; siehe Bilder 27 und 27a:

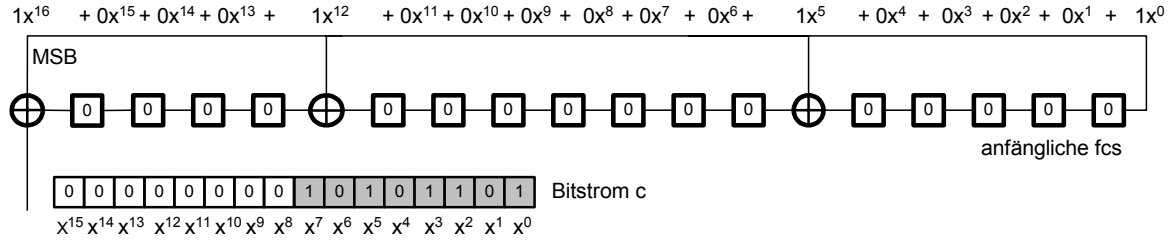


Bild 27: Bitstrom c vor dem Linksschieben

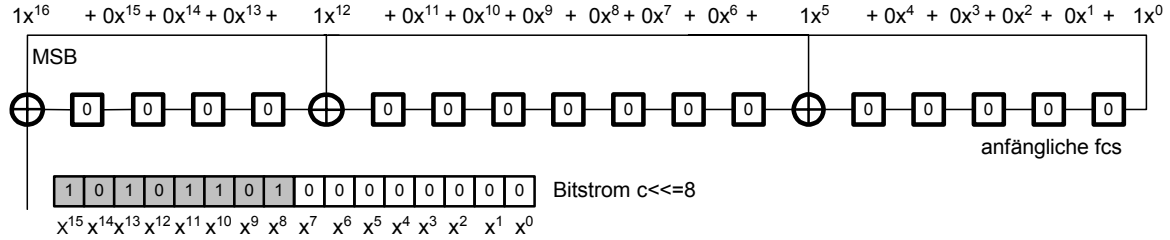


Bild 27a: Bitstrom c nach dem Linksschieben

Die Funktion crc setzt ihre Arbeit fort, und bearbeitet in der for-Schleife **for(i=0;i<8;i++)** Bit für Bit die Variable c. Für jedes Bit ermittelt sie zunächst das MSB: **(fcs^c) & 0x8000**

Beispiel gemäß Situation nach Bild 27a:

1,

fcs = 0000 0000 0000 0000
^c = **1010 1101** 0000 0000
fcs^c = 1010 1101 0000 0000

2,

fcs^c = 1010 1101 0000 0000
& 8000 = 1000 0000 0000 0000
(fcs^c) & 8000 = **1**000 0000 0000 0000

Ergebnis: MSB=**1**

Die crc-Simulationssoftware liefert auf ihre Weise das gleiche Ergebnis wie die modulo2-Divisionshardware. Vergleiche mit Bild 27b.

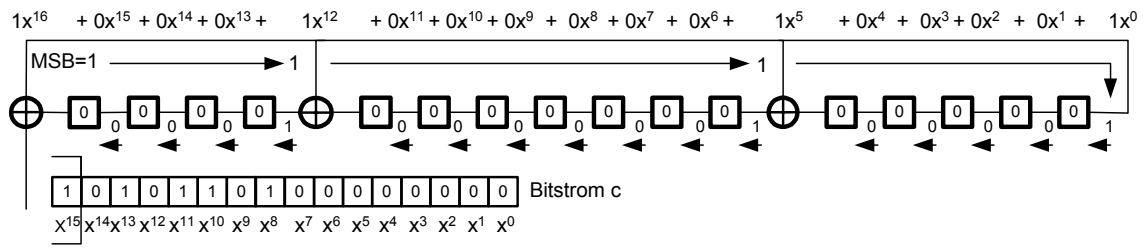


Bild 27b: Die modulo2 Divisions-Hardware liefert MSB=1

Die Divisions-Hardware schließt den ersten Schritt ab, und liefert durch Linksschieben der neuen Informationen das erste temporäre fcs: Siehe Bild 27c.

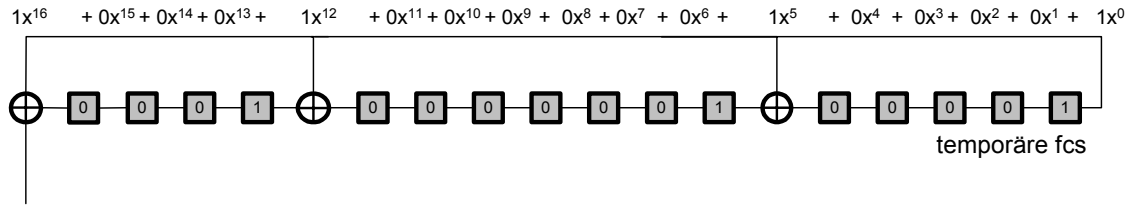


Bild 27c: Temporäre fcs nach dem ersten Schritt

Zum gleichen Ergebnis kommt die crc-Simulationssoftware. Wegen MSB=1 folgt:
fcs = (fcs<<1)^mask; (vergleiche mit Bild 26)

3, fcs = 0000 0000 0000 0000

fcs<<1 = 0000 0000 0000 0000
^mask = 0001 0000 0010 0001
fcs = 0001 0000 0010 0001

Die Funktion crc setzt ihre Arbeit fort, und bearbeitet nach **c<<=1** das nächste Bit im Bitstrom. Vergleiche mit Bild 26.

Die modulo2 Divisions-Hardware bearbeitet ebenfalls das nächste Bit im Bitstrom, nämlich $0x^{14}$, und liefert MSB=0. Siehe Bild27d.

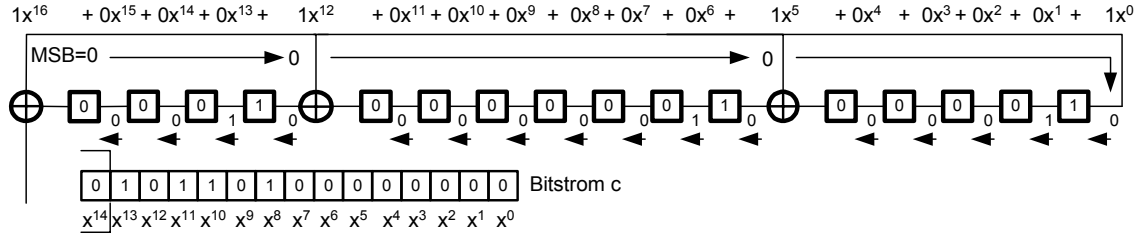


Bild 27d: Die modulo2 Divisions-Hardware liefert MSB=0

Durch das übliche Linksschieben um eine Bitposition kommt sie zur nächsten temporären fcs. Siehe Bild 27e.

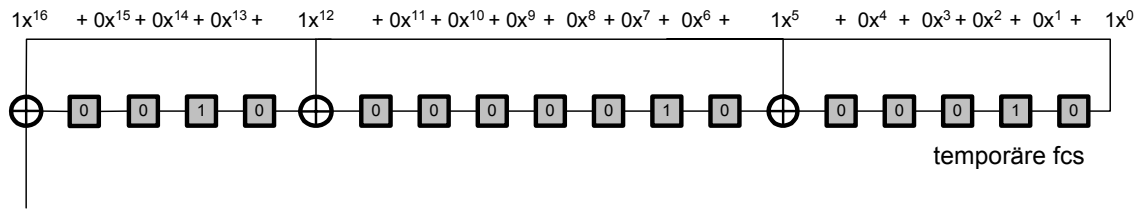


Bild 27e: Temporäre fcs nach dem zweiten Schritt

Die crc-Simulationssoftware kommt zu dem gleichen Ergebnis.

Gemäß Situation nach Bild 27d gilt:

1, fcs = 0001 0000 0010 0001
^c = **0101 1010** 0000 0000
fcs^c = 0100 1010 0010 0001

2, fcs^c = 0100 1010 0010 0001
& 8000 = 1000 0000 0000 0000
(fcs^c) & 8000 = **0**000 0000 0000 0000

Ergebnis: MSB=0

Wegen MSB=0 folgt: **fcs<<1**; (vergleiche mit Bild 26)

3, fcs = 0001 0000 0010 0001

fcs<<1 = **0010 0000 0100 0010**

Die Funktion crc setzt ihre Arbeit fort, und bearbeitet nach c<<=1 das nächste Bit im Bitstrom. Dies geschieht solange bis die for-Schleife beendet ist. Danach liefert crc die endgültige fcs für 1 Oktet**: return fcs;** (vergleiche Bild 26).

1.3.1.3.8 fcs-Berechnung für n Oktets

Wir erinnern uns, die Receive Machine empfängt die MAC PDU und speichert sie im rxm_puffer[14]. In welcher zeitlichen Reihenfolge kommen die Oktets beim Empfänger an? Sie werden von der Transmit Machine des Senders in das LAN eingespeist, und zwar zuerst Oktet 0 und zuletzt Oktet 13. Also kommen sie auch in dieser Reihenfolge beim Empfänger an. Siehe Bild 28.

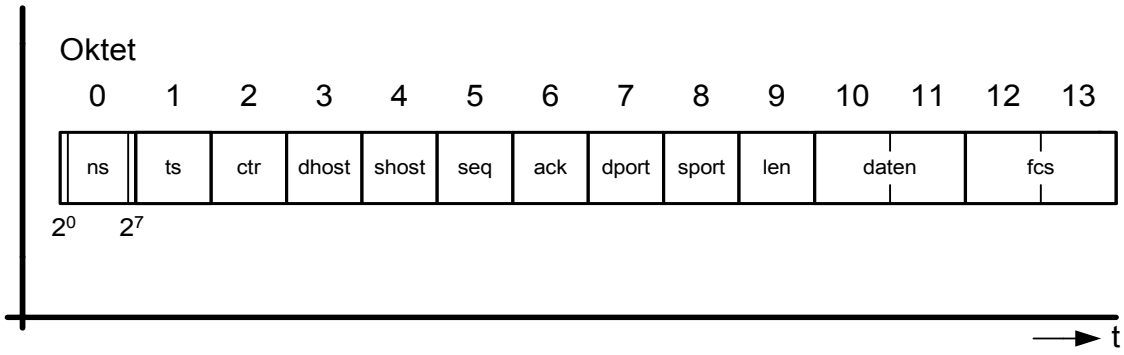


Bild 28: Zeitliches Eintreffen der Oktets beim Empfänger

Benutzt die Receive Machine für die fcs-Berechnung die modulo2 Divisions-Hardware, dann bedeutet dies, daß sie die individuellen Oktets der Hardware in der gleichen Reihenfolge zuführen muß. Also zuerst Oktet 0 und zuletzt Oktet 13. Mit anderen Worten, die MAC PDU stellt sich als Bitstrom mit 14•8 Bits = 112 Bits dar, wobei Bit 2^7 in Oktet 0 die höchstwertige Stelle 2^{111} einnimmt, und Bit 2^0 in Oktet 13 die niederwertigste Stelle 2^0.

Für die Ermittlung der MSBs für n=14 Oktets wird deshalb mit dem höchstwertigen Bit 2^{111} begonnen und mit dem niederwertigsten Bit 2^0 aufgehört. So ist es einsichtig, warum das High-Byte der fcs im niederwertigeren Oktet 12 und das Low-Byte der fcs im höherwertigeren Oktet 13 der MAC PDU hinterlegt ist. Siehe Bild 29.
Anmerkung: Die fcs ≠ 0 ermittelt die Transmit Machine des Senders und speichert die High- und Low-Bytes der fcs in den richtigen Oktets 12 und 13.

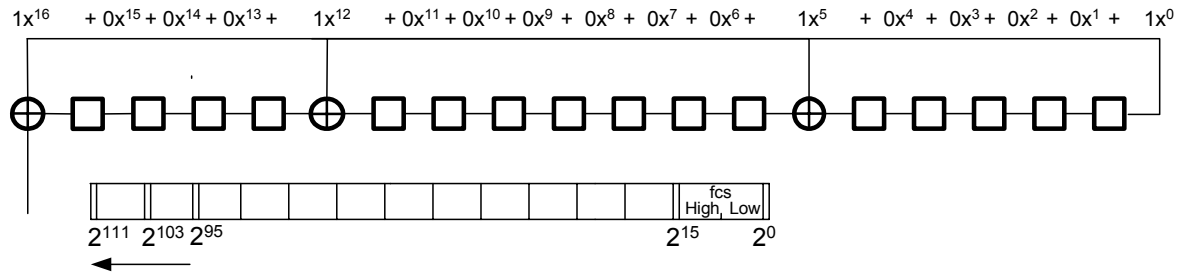


Bild 29: Bitstrom für n=14 Oktets

Die modulo2 Divisions-Hardware beginnt ihre Arbeit mit dem anfänglichen Schieberegisterinhalt fcs=0 und ermittelt ab Bit 2^{111} des Bitstroms

1, für Oktet 0 **fcs_0**.

Danach bearbeitet sie Oktet 1, und benutzt als anfänglichen Wert den derzeitigen Schieberegisterinhalt fcs_0. Sie ermittelt ab Bit 2^{103} des Bitstroms

2, für Oktet 0 und Oktet 1 **fcs_1**.

Anschließend bearbeitet sie Oktet 2, und benutzt als anfänglichen Wert den derzeitigen Schieberegisterinhalt fcs_1. Sie ermittelt ab Bit 2^{95} des Bitstroms

3, für Oktet 0, Oktet 1 und Oktet 2 **fcs_2** usw.

Nach der Bearbeitung von 14 Oktets ist schließlich die endgültige fcs für die MAC PDU bestimmt und hoffentlich 0.
Der beschriebene Algorithmus ist leicht durch Software zu simulieren. Es ist lediglich die Funktion crc in eine for-Schleife zu „packen“. Siehe Bild 30.

```
fcs=0;                                          /*  fcs berechnen   */
for(i=0;i<rxm_max;i++)
   fcs=crc(fcs,rxm_puffer[i],mask);
```

Bild 30: fcs-Berechnung für 14 Oktets

Das gleiche Simulations-Software benutzt die Receive Machine (vergleiche mit Bild 11). Dort ist mit **#define rxm_max 14** die Anzahl der Oktets=14 bestimmt. Oktet für Oktet, beginnend mit Oktet 0 (i=0), arbeitet die Simulations-Software den Bitstrom ab, und liefert am Ende die fcs der MAC PDU.

1.3.1.4 Fehlerauswertung und Botschaften

Die Receive Machine identifiziert zunächst zwei verschiedene Ereignisse: Ein Frame ist unbeschädigt eingetroffen oder beschädigt. Im ersten Fall bekommt der Frame die Kennzeichnung **ok**, und die Receive Machine identifiziert ihn zusätzlich als Daten- oder Token-Frame. Im zweiten Fall bekommt der Frame die Kennzeichnung **io_error**, und eine weitere Identifizierung ist hinfällig. Wer ist an diesen Informationen interessiert? Es sind die Access Control Machine und die Interface Machine.
Die Access Control Machine möchte über alle Ereignisse informiert werden, also über die Ankunft

- eines Daten_Frames,
- eines Token-Frames,
- eines Claim-Frames oder
- eines Error-Frames.

Die Interface Machine ist nur an der Ankunft

- eines Daten-Frames interessiert.

Nachdem die Receive Machine die fcs-Berechnung beendet hat, führt sie das folgende Programm-Fragment aus. Vergleiche auch mit Bild 11: Betriebssystem-Funktion RxM_exe.

```
switch(fcs)
{
 case 0:
  switch(rxm_puffer[ctrl] & 0x48)
  {
   case data:                                    /*   data empfangen   */
         llc_event=ok;
         sende_llc_event(llc_event,rxm_puffer);          /* an ifm */

         acm_event=ok;
         sende_acm_event(acm_event,rxm_puffer);          /* an acm */
   break;

   case token:                                   /*   token empfangen  */
         acm_event=ok;
         sende_acm_event(acm_event,rxm_puffer);          /* an acm */
   break;

   case claim:                                   /*   claim empfangen  */
         acm_event=ok;
         sende_acm_event(acm_event,rxm_puffer);          /* an acm */
   break;
  }
 break;
 default:                                              /*   fcs-Error      */
        acm_event=io_error;
        sende_acm_event(acm_event,rxm_puffer);    /*     an acm      */
 break;
}
```

Bild 31: fcs-Auswertung

Mit switch(fcs) stellt sie fest, ob fcs=0 oder fcs≠0 ist.

1, Ist **fcs=0** (case 0), überprüft die Receive Machine das Control-Feld der empfangenen MAC PDU: rxm_puffer[ctrl] = rxm_puffer[2], wegen #define ctrl 2.

- Ist rxm_puffer[ctrl] = 0x40 (data), handelt es sich um einen Daten-Frame.
- Ist rxm_puffer[ctrl] = 0x50 (data), handelt es sich um einen Response-Frame.
- Ist rxm_puffer[ctrl] = 0x08 (token), handelt es sich um einen Token-Frame.
- Ist rxm_puffer[ctrl] = 0x00 (claim), handelt es sich um einen Claim-Frame.

Durch die UNDierung mit 0x48 sind alle drei Frame-Typen identifizierbar, wobei der Response-Frame (0x50) wie ein Daten-Frame behandelt wird. **Anmerkung**: Welche Rolle der Response-Frame im MAC-Layer spielt, werden wir bei der Behandlung der Access Control Machine sehen.
Hat die Receive Machine einen Daten-Frame empfangen hat, schickt sie zwei Botschaften ab, und zwar eine **ok**-Botschaft über die Interface Machine an den Logical Link Control-Layer 2b (LLC-Layer 2b) und eine **ok**-Botschaft an die Access Control Machine des MAC-Layers.
Hat die Receive Machine einen Token Frame empfangen, schickt sie nur eine **ok**-Botschaft an die Access Control Machine. Der LLC-Layer interessiert sich dafür nicht.
Die Receive Machine speichert zunächst die Botschaft in den beiden statischen char-Variablen **llc_event** und **acm_event**, fügt die Adresse des **rxm_puffers** hinzu, und veranlaßt anschließend den Transport dieser Informationen. Sie benutzt zu diesem Zweck die zwei Service Access Points (SAPs)

- **sende_llc_event(llc_event,rxm_puffer)**; Botschaft an die Interface Machine
- **sende_acm_event(acm_event,rxm_puffer)**; Botschaft an die Access Control Machine

2, Ist **fcs≠0** (default), schickt die Receive Machine eine **io_error**-Botschaft an die Access Control Machine. Sie speichert zunächst das Ereignis in der Variablen acm_event, und benutzt für den anschließenden Transport den entsprechenden SAP.

Hat die hardware-basierte Fehlererkennung des UART einen Fehler angezeigt, schickt die Receive Machine ebenfalls eine **io_error**-Botschaft an die Access Control Machine. Siehe folgendes Programm-Fragment in Bild 32 und vergleiche mit Bild 11.

```
                                                    /* Ov-, Par-, Fra-Error  */
case 0xc6: inbyte(lsr1);                          /* clear line status-interrupt */
        outbyte(fcr1,0xcb);                          /*Receiver-FIFO Reset */

        acm_event = io_error;
        sende_acm_event(acm_event,rxm_puffer);            /*     an acm     */
break;
                                                     /* Character timeout   */

case 0xcc: inbyte(rbr1);                          /* clear timeout-interrupt */
        outbyte(fcr1,0xcb);                          /*Receiver-FIFO Reset */

        acm_event = io_error;
        sende_acm_event(acm_event,rxm_puffer);            /*     an acm     */
break;
```

Bild 32: Auswertung der UART-Fehlermeldungen

Wie das Korrespondenz-Schema zwischen der Receive Machine, der Interface Machine und der Acces Control Machine aussieht, illustriert das folgende Bild 33.

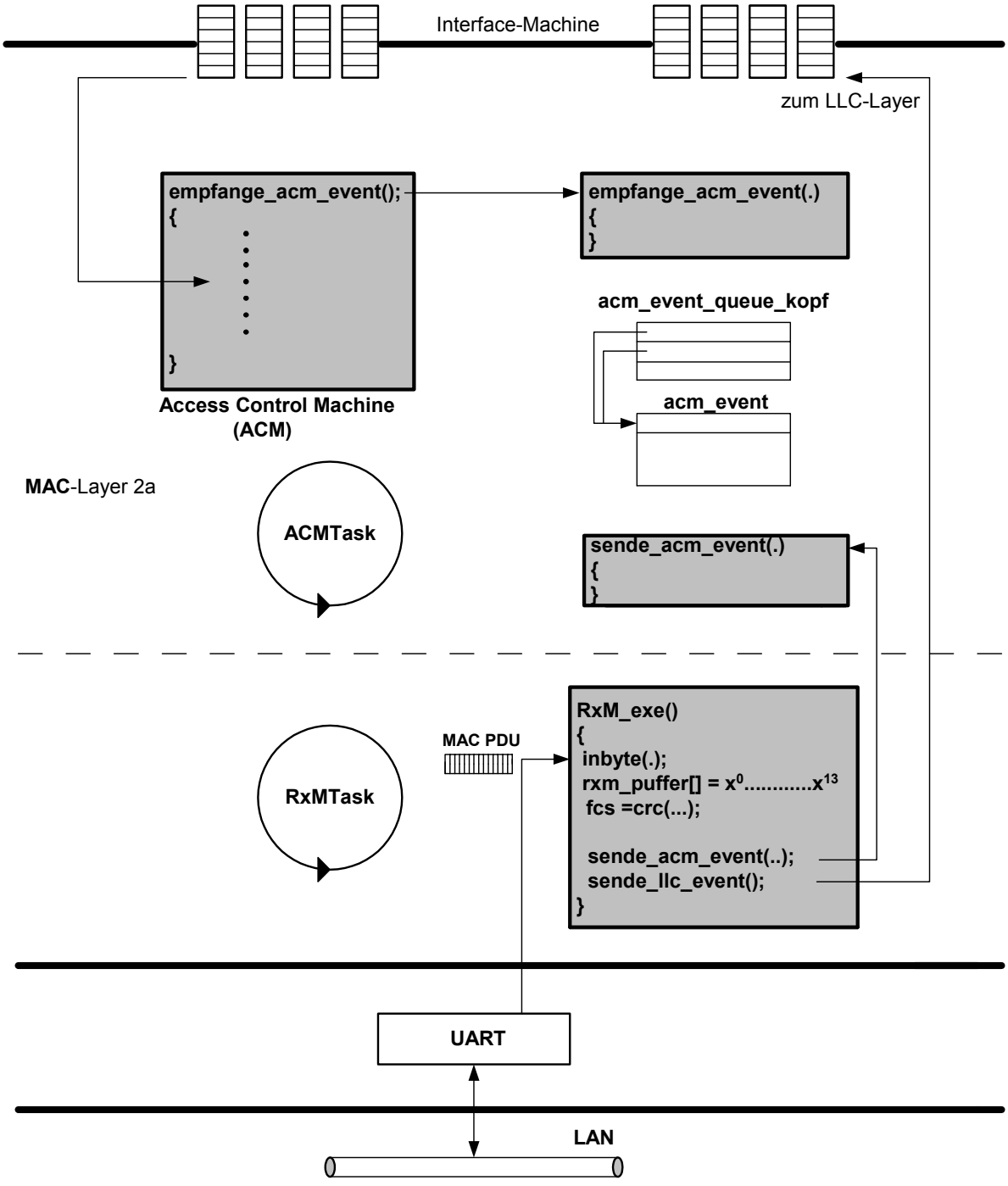


Bild 33: Korrespondenz zwischen Receive Machine, Access Control Machine und Interface Machine

1.3.1.4.1 Botschaften an die Interface Machine

Betrachten wir als erstes die Interface Machine. Sie ist in zwei logische Funktionseinheiten unterteilt: Eine Empfangseinheit (im Bild 33 rechts) und eine Sendeeinheit (im Bild 33 links). Jeder Rechner erhält während des Betriebs immer wieder das Token und ist damit sendeberechtigt. Hat er das Token nicht, arbeitet er im Empfangsmodus. Nur in diesem Modus ist die Receive Machine aktiv. Identifiziert sie eine empfangene MAC PDU als Daten-Frame, extrahiert sie das fcs-Feld und den MAC Header H3 (ns, ts, ctrl) und sendet mittels sende_llc_event(..); den Rest als LLC PDU an die Empfangseinheit der Interface Machine. Von dort kann dann der LLC Layer mittels empfange_llc_event(.); die LLC PDU abholen und das Sicherungsprotokoll starten. Die Empfangseinheit der Interface Machine wir also benutzt, um die LLC PDU vom MAC Layer zum LLC Layer weiterzuleiten. Siehe Bild 34.

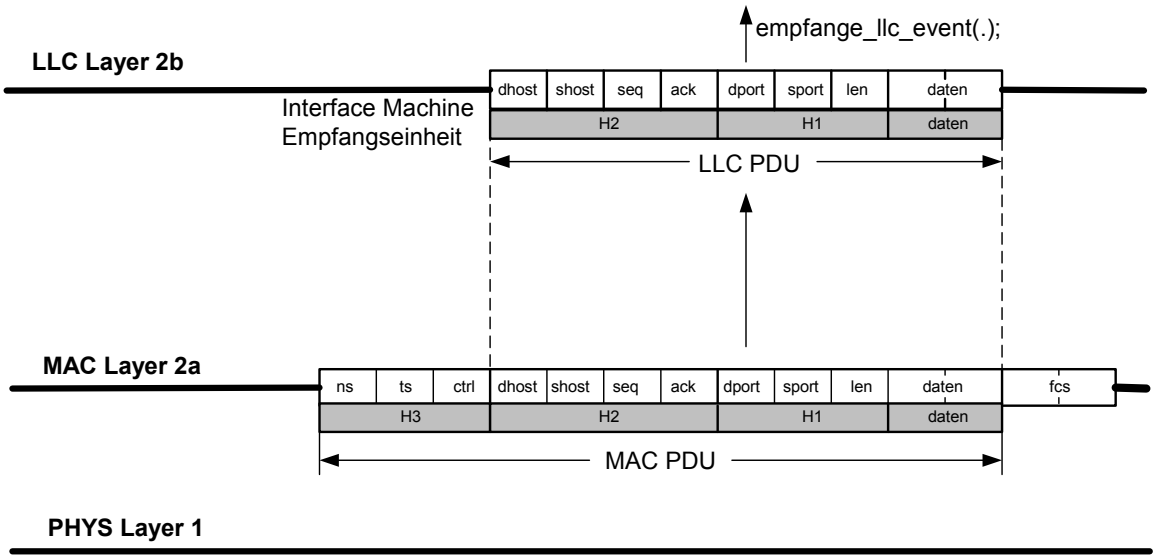


Bild 34: Weiterleiten der LLC PDU vom MAC Layer zum LLC Layer

Die Empfangseinheit ist als ein array von Mailboxen realisiert, in die die LLC PDUs eingekettet werden. Die Idee ist, in jedem Rechner (host_id) für jeden Quell-Rechner (shost) eine individuelle Mailbox zur Verfügung zu stellen. Auf diese Weise kann jeder Ziel-Rechner (dhost) erkennen, von welchem Quell-Rechner eine LLC PDU gekommen ist. Der Ziel-Rechner ist jetzt in der Lage mit dem Quell-Rechner ein Sicherungsprotokoll auszuführen. Betrachten wir dazu einige Beispiele. Siehe Bilder 35a und 35b.

Beim Systemstart gibt der LLC Layer seinem Rechner eine individuelle Identifikationsnummer: host_id = 0...n (z.B. n=3). Diese Nummer ist nicht zu verwechseln mit der Stationsnummer, die von der MAC-Schicht vergeben wird. Jede Mailbox in der Empfangseinheit der Interface Machine ist ebenfalls durch eine individuelle Nummer im Bereich von 0...n gekennzeichnet. Über die gleichlautenden Nummern ist jeder shost im dhost mit ´seiner´ Mailbox verbunden. Mit anderen Worten, jeder shost hat in jedem dhost eine ihm zugeordnete Mailbox, in die seine LLC PDU eingekettet wird.

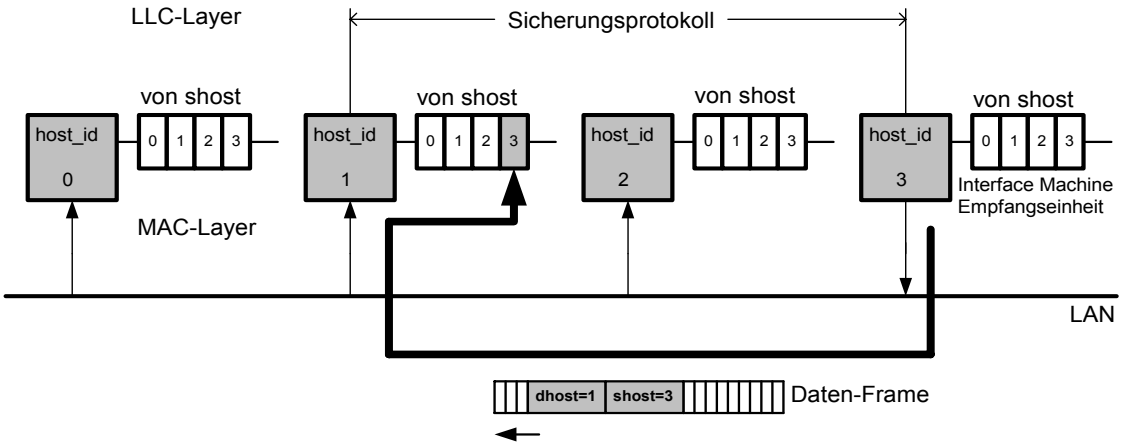


Bild 35a: Rechner 1 empfängt LLC PDU von Rechner 3

Im Bild 35a ist der Rechner 3 (host_id = 3) sendeberechtigt. Alle anderen arbeiten im Empfangsmodus. Rechner 3 (shost = 3) schickt an Rechner 1 (dhost = 1) einen Daten-Frame. Die dortige Receive Machine kettet ihn als LLC PDU in die Mailbox 3 ein. Die LLC-Layer in Rechner 1 und Rechner 3 führen daraufhin ein Sicherungsprotokoll aus.

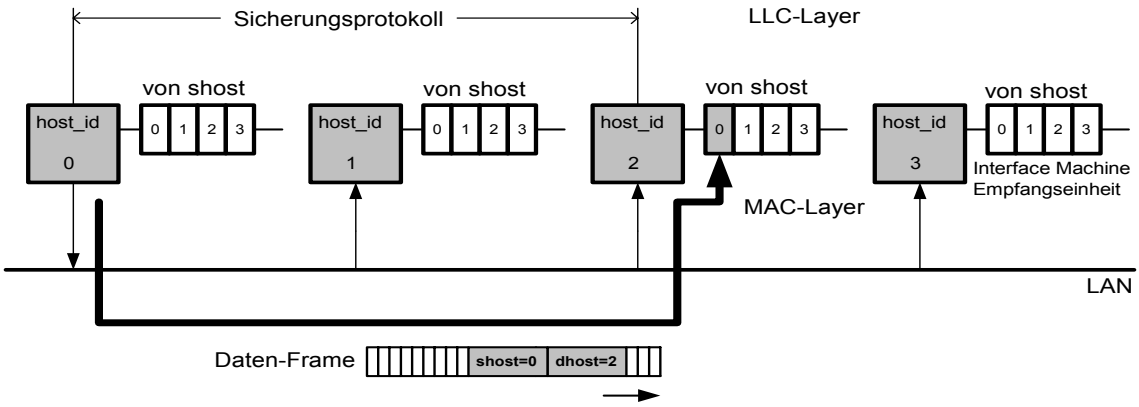


Bild 35b: Rechner 2 empfängt LLC PDU von Rechner 0

Im Bild 35b ist der Rechner 0 (host_id = 0) sendeberechtigt. Alle anderen arbeiten im Empfangsmodus. Rechner 0 (shost = 0) schickt an Rechner 2 (dhost = 2) einen Daten-Frame. Die dortige Receive Machine kettet ihn als LLC PDU in die Mailbox 0 ein. Die LLC-Layer in Rechner 0 und Rechner 2 führen daraufhin ein Sicherungsprotokoll aus.

Für die Übertragung der LLC PDUs an den Empfangsteil der Interface Machine benutzt die Receive Machine die folgende Funktion. Siehe Bild 36.

```
#include<struct6.h>

extern llc_event_struct      llc0_event,llc1_event,llc2_event,llc3_event;
extern unsigned char         host_id;

void sende_llc_event(char event,unsigned char rxm_puffer[])
{
 if(rxm_puffer[3] == host_id)
   {
   switch((rxm_puffer[4] & 0x7f))
   {
    case 0: llc0_event.event    = event;
          llc0_event.dhost      = rxm_puffer[3];
          llc0_event.shost      = rxm_puffer[4];
          llc0_event.seq        = rxm_puffer[5];
          llc0_event.ack        = rxm_puffer[6];
          llc0_event.dport      = rxm_puffer[7];
          llc0_event.sport      = rxm_puffer[8];
          llc0_event.len        = rxm_puffer[9];
          llc0_event.daten[0]   = rxm_puffer[10];
          llc0_event.daten[1]   = rxm_puffer[11];
          sende_llc0_event(&llc0_event);
    break;
    case 1: llc1_event.event    = event;
          llc1_event.dhost      = rxm_puffer[3];
          llc1_event.shost      = rxm_puffer[4];
          llc1_event.seq        = rxm_puffer[5];
          llc1_event.ack        = rxm_puffer[6];
          llc1_event.dport      = rxm_puffer[7];
          llc1_event.sport      = rxm_puffer[8];
          llc1_event.len        = rxm_puffer[9];
          llc1_event.daten[0]   = rxm_puffer[10];
          llc1_event.daten[1]   = rxm_puffer[11];
          sende_llc1_event(&llc1_event);
    break;
    case 2: llc2_event.event    = event;
          llc2_event.dhost      = rxm_puffer[3];
          llc2_event.shost      = rxm_puffer[4];
          llc2_event.seq        = rxm_puffer[5];
          llc2_event.ack        = rxm_puffer[6];
          llc2_event.dport      = rxm_puffer[7];
          llc2_event.sport      = rxm_puffer[8];
          llc2_event.len        = rxm_puffer[9];
          llc2_event.daten[0]   = rxm_puffer[10];
          llc2_event.daten[1]   = rxm_puffer[11];
          sende_llc2_event(&llc2_event);
    break;
```

```
    case 3: llc3_event.event    = event;
            llc3_event.dhost    = rxm_puffer[3];
            llc3_event.shost    = rxm_puffer[4];
            llc3_event.seq      = rxm_puffer[5];
            llc3_event.ack      = rxm_puffer[6];
            llc3_event.dport    = rxm_puffer[7];
            llc3_event.sport    = rxm_puffer[8];
            llc3_event.len      = rxm_puffer[9];
            llc3_event.daten[0] = rxm_puffer[10];
            llc3_event.daten[1] = rxm_puffer[11];
            sende_llc3_event(&llc3_event);
    break;
   }
  }
}
```

Bild 36: Betriebssystem-Funktion sende_llc_event (Datei sendllc.c)

Die Funktion wird mit zwei Parametern aufgerufen: event und rxm_puffer[].
Mit **if(rxm_puffer[3] == host_id)** überprüft sie als erstes, ob der angekommene Daten-Frame für den Ziel-Rechner bestimmt ist. Zur Erinnerung: rxm_puffer[3] entspricht rxm_puffer[dhost]. Warum ist diese Überprüfung notwendig? Betrachten wir dazu das folgende Bild 37:

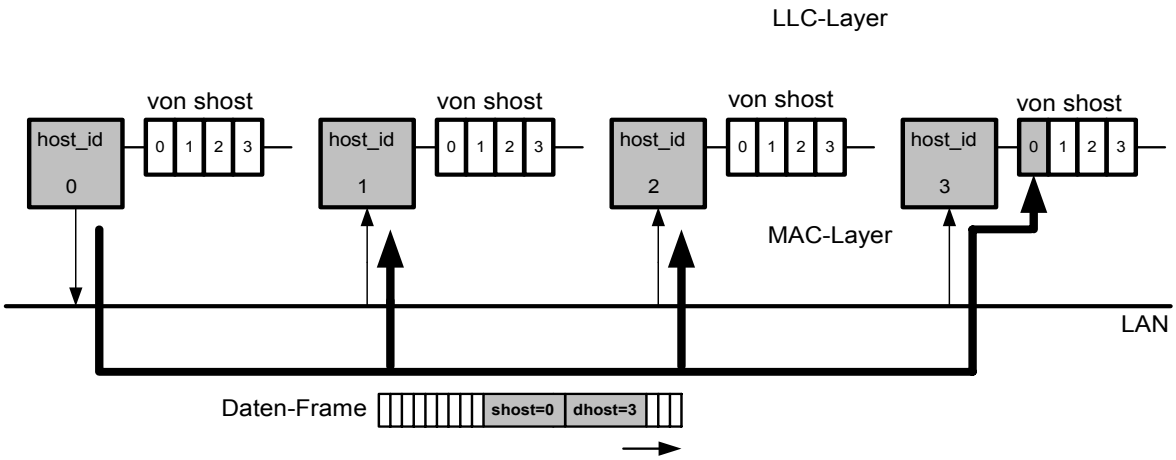


Bild 37: Rechner 1,2 und 3 ´hören´am LAN mit

Hier ist der Rechner 0 sendeberechtigt; alle anderen Rechner befinden sich im Empfangsmodus.
Rechner 0 schickt einen Daten-Frame ins LAN, der für Rechner 3 bestimmt ist. Doch weil Rechner 1, 2 und 3 im Empfangsmodus arbeiten, nehmen die Receive Machines aller drei Rechner den Daten-Frame auf; sie ´hören´ am LAN mit. Aufgrund des Auswertungsergebnisses leitet nur Rechner 3 die LLC PDU an ihre Interface Machine weiter, während Rechner 1 und 2 die LLC PDU ´verwerfen´.

Hat sich einer der Rechner als Empfänger erkannt, muß er als nächstes den Absender, also den shost, feststellen. Er tut dies mit **switch((rxm_puffer[4] & 0x7f))**. Zur Erinnerung: rxm_puffer[4] entspricht rxm_puffer[shost]. Und er bekommt für shost (ausgeschlossen ist die eigene host_id-Nummer) eine der verbleibenden Nummern 0,1,2 oder 3. So kann er anschließend die empfangene LLC PDU in die Mailbox 0,1,2 oder 3 seiner Interface Machine einketten.
Warum wird rxm_puffer[4] & 0x7f ausgeführt, also Bit 2^7 in rxm_puffer[shost] ausgeblendet? Es ist ein **C**ontrol-Bit (C-Bit) und hat den Wert 0 oder 1. Es ist ausschließlich für den LLC Layer von Interesse. Jede LLC PDU oder auch eine Anzahl von LLC PDUs, die ein shost an einen dhost schickt, wird vom dhost quittiert.

- Trägt ein Daten-Frame gültige Daten, ist Bit 2^7 im shost-Feld **0**.
- Trägt ein Daten-Frame **keine** gültigen Daten, sondern wird nur benutzt, um eine vorher empfangene LLC PDU zu quittieren, ist Bit 2^7 im shost-Feld **1**.

Betrachten wir dazu ein Beispiel. Siehe Bilder 38 und 38a.

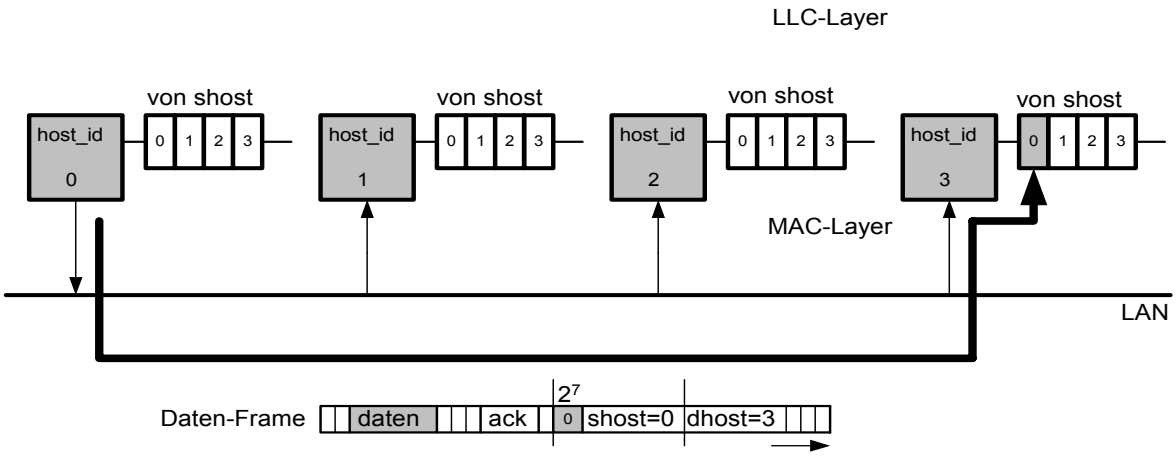


Bild 38: Rechner 0 schickt Rechner 3 gültige Daten

Hier schickt der Quell-Rechner 0 (shost=0) an den Ziel-Rechner 3 (dhost=3) einen Daten-Frame mit gültigen Daten im **daten**-Feld. Das C-Bit im shost-Feld ist daher 0.

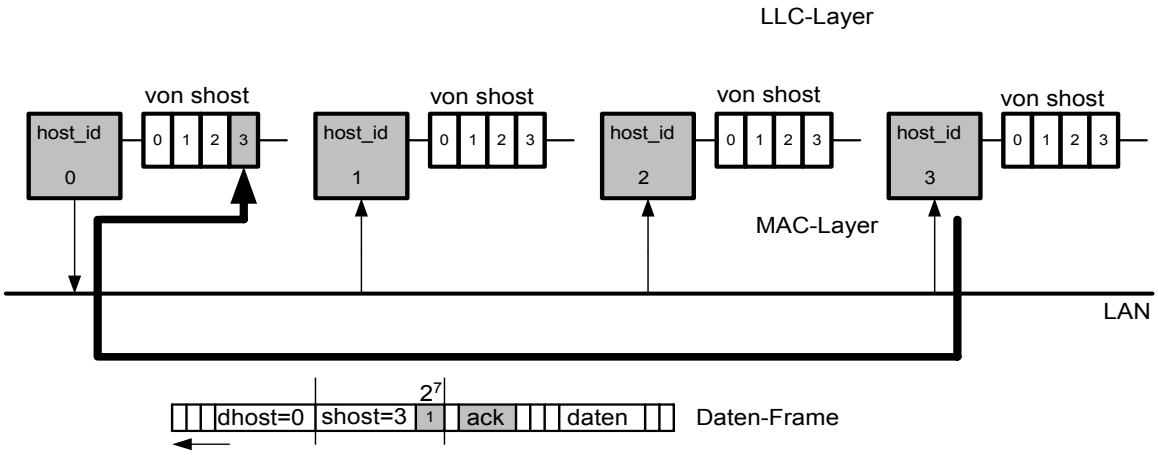


Bild 38a: Rechner 3 schickt Rechner 0 eine Quittung

Der vorherige Ziel-Rechner (dhost=3) wird zum Quell-Rechner (shost=3), und der vorherige Quell-Rechner (shost=0) wird zum Ziel-Rechner (dhost=0). Der jetzige Quell-Rechner 3 schickt an den jetzigen Ziel-Rechner 0 einen Daten-Frame zurück. Er trägt **keine** gültigen Daten im daten-Feld, und ist daher ein purer **Quittungs**-Frame. Aus diesem Grund hat das C-Bit im shost-Feld den Wert 1. Von Interesse ist jetzt das **ack**-Feld. Es wird vom Sicherungsprotokoll im LLC Layer des Ziel-Rechners 0 ausgewertet.
Nachdem die Betriebssystem-Funktion sende_llc_event(..) den shost identifiziert hat, kettet sie die empfangene LLC PDU in die korrespondierende Mailbox der Interface Machine ein. Sie benutzt dabei die folgenden Funktionen (vergleiche mit Bild 36):

- wenn shost = 0 (case 0:) dann **sende_llc0_event(&llc0_event);**
- wenn shost = 1 (case 1:) dann **sende_llc1_event(&llc1_event);**
- wenn shost = 2 (case 2:) dann **sende_llc2_event(&llc2_event);**
- wenn shost = 3 (case 3:) dann **sende_llc3_event(&llc3_event);**

Jede Mailbox ist eine Queue und besteht aus einem **llc_event_queue_kopf** und einem llc_event-Element. Die Struktur dieser Queue ist durch die folgenden zwei Datentypen definiert:

```
typedef struct s13
        {
          struct s14 *erster_der_queue;
          struct s14 *letzter_der_queue;
          short int   anzahl_der_llc_events;
        } llc_event_queue_kopf_struct;

typedef struct s14
        {
          struct s14 *naechster_in_der_queue;
          char              event;
          unsigned char dhost;
          unsigned char shost;
          unsigned char seq;
          unsigned char ack;
          unsigned char dport;
          unsigned char sport;
          unsigned char len;
          unsigned char daten[2];
        } llc_event_struct;
```

Bild 39: Definition der Datentypen llc_event_queue_kopf_struct und llc_event_struct

Die Datentypen heißen **s13** und **s14**. Beide sind umbenannt, und zwar s13 in llc_event_queue_kopf_struct und s14 in llc_event_struct.

- Eine Variable des Typs llc_event_queue_kopf_struct repräsentiert den **shost = 0..3** durch llc_event_queue_kopf[0]..llc_event_queue_kopf[3] und
- eine Variable des Typs llc_event_struct repräsentiert das **Ereignis** durch char event und die **LLC PDU** durch unsigned char dhost...unsigned char daten[2].

Die beiden Zeiger erster_der_queue und letzter_der_queue im llc_event_queue_kopf zeigen auf das erste und letzte llc_event-Element in der Queue. Die short int-Variable anzahl_der_llc_events zählt die Anzahl der eingeketteten llc_event-Elemente, und der Zeiger naechster_in_der_queue in einem llc_event-Element zeigt auf das nachfolgende Element. Damit das aufgetretene Ereignis zusammen mit der empfangenen LLC PDU in eine der Queues eingekettet werden kann, muß das Betriebssystem die erforderliche Anzahl von Queues zur Verfügung stellen. So installiert unmittelbar nach dem Systemstart die Betriebssystem-Funktion init für jeden shost einen **llc_event_queue_kopf**, und für jede Queue ein **llc_event**-Element. Siehe Bild 40.

```
#define null (void*)       0
#define llc_event_max   4
                                •
                                •
#include<struct6.h>
                                •
                                •
llc_event_queue_kopf_struct llc_event_queue_kopf[llc_event_max];
llc_event_struct          llc0_event;
llc_event_struct          llc1_event;
llc_event_struct          llc2_event;
llc_event_struct          llc3_event;
                                •
                                •
void init(void)
{
                                •
                                •
            /*  4 llc_event_queue_Koepfe installieren */

 for(i=0;i<llc_event_max;i++)
   {
    llc_event_queue_kopf[i].erster_der_queue        = null;
    llc_event_queue_kopf[i].letzter_der_queue       = null;
    llc_event_queue_kopf[i].anzahl_der_llc_events  = 0;
   }

          /* 1 llc0_event-Element installieren */

llc0_event.naechster_in_der_queue      = null;
llc0_event.event                        = 0;
llc0_event.dhost                        = 0;
llc0_event.shost                        = 0;
llc0_event.seq                          = 0;
llc0_event.ack                          = 0;
llc0_event.dport                        = 0;
llc0_event.sport                        = 0;
llc0_event.len                          = 0;
llc0_event.daten[0]                     = 0;
llc0_event.daten[1]                     = 0;


          /* 1 llc1_event-Element installieren */

llc1_event.naechster_in_der_queue      = null;
llc1_event.event                       = 0;
llc1_event.dhost                       = 0;
llc1_event.shost                       = 0;
llc1_event.seq                         = 0;
llc1_event.ack                         = 0;
llc1_event.dport                       = 0;
llc1_event.sport                       = 0;
llc1_event.len                         = 0;
llc1_event.daten[0]                    = 0;
llc1_event.daten[1]                    = 0;
```

```
        /* 1 llc2_event-Element installieren */

llc2_event.naechster_in_der_queue      = null;
llc2_event.event                       = 0;
llc2_event.dhost                       = 0;
llc2_event.shost                       = 0;
llc2_event.ack                         = 0;
llc2_event.dport                       = 0;
llc2_event.sport                       = 0;
llc2_event.len                         = 0;
llc2_event.daten[0]                    = 0;
llc2_event.daten[1]                    = 0;

        /* 1 llc3_event-Element installieren */

llc3_event.naechster_in_der_queue      = null;
llc3_event.event                       = 0;
llc3_event.dhost                       = 0;
llc3_event.shost                       = 0;
llc3_event.seq                         = 0;
llc3_event.ack                         = 0;
llc3_event.dport                       = 0;
llc3_event.sport                       = 0;
llc3_event.len                         = 0;
llc3_event.daten[0]                    = 0;
llc3_event.daten[1]                    = 0;
                              •
                              •
}
```

Bild 40: Fragment der Betriebssystem-Funktion init (Datei init.c)

Die Empfangseinheit der Interface-Machine gestaltet sich danach wie folgt. Siehe Bild 41.

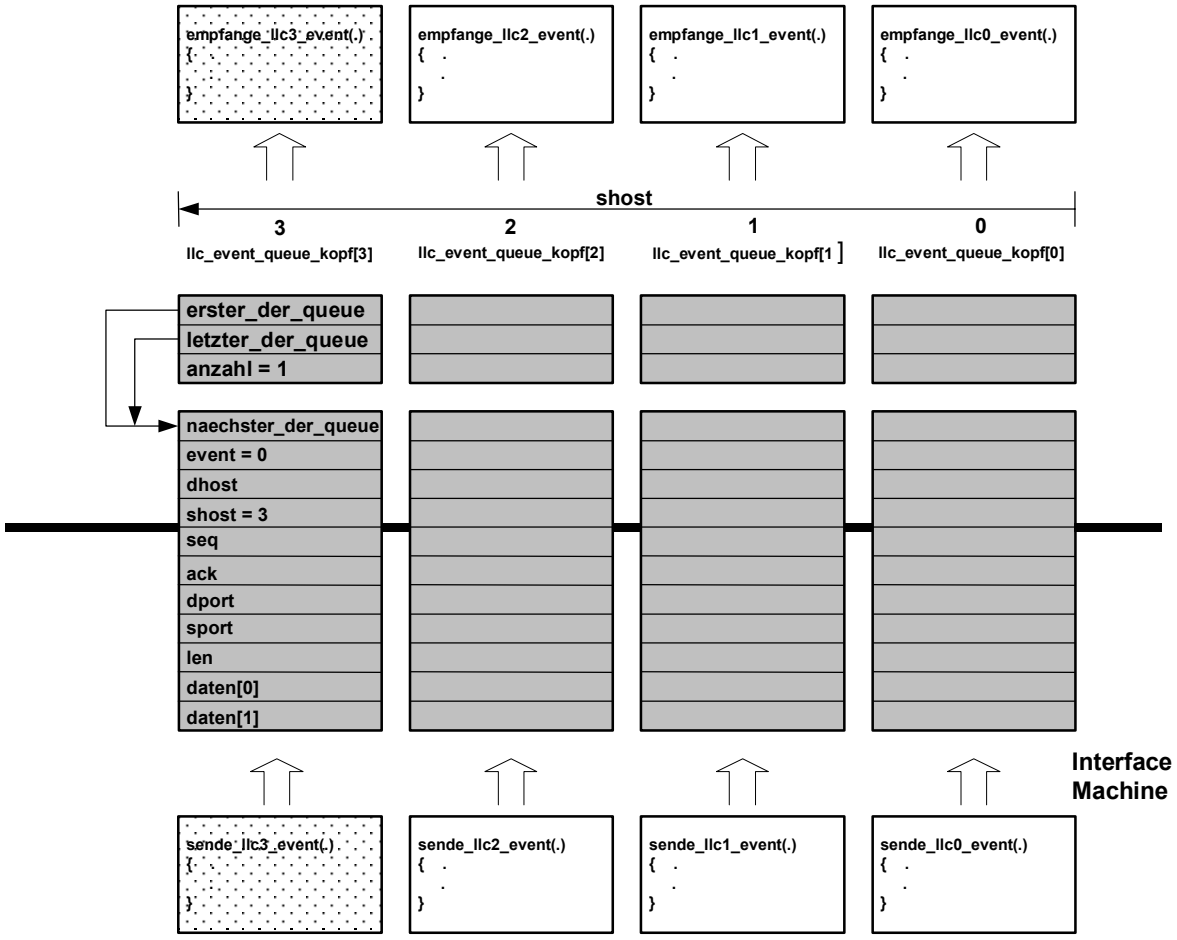


Bild 41: Empfangseinheit der Interface-Machine

Das Bild demonstriert zusätzlich den Fall, daß die Receive Machine **shost=3** als Quell-Rechner identifiziert hat. Sie überträgt **event=0** (ok) und **rxm_puffer[3]..rxm_puffer[11]** (LLC PDU) in das **llc3_event**-Element, und ruft anschließend die Funktion sende_llc3_event(&llc3_event) auf. Diese Funktion kettet das llc3_event-Element (LLC PDU) in den **llc_event_queue_kopf[3]** (shost=3) ein, und bekommt deshalb als Parameter die Adresse des llc3_event-Elements. Die analoge Prozedur geschieht, wenn andere shosts als Quell-Rechner identifiziert worden sind (vergleiche mit Bild 36).
Auf der anderen Seite der Interface Machine kann jetzt der LLC Layer mittels der Funktionen

- **empfange_llc0_event(.);**
- **empfange_llc1_event(.);**
- **empfange_llc2_event(.);**
- **empfange_llc3_event(.);**

die eingeketteten llc_event-Elemente ausketten und die entsprechenden Sicherungsprotokolle starten.

Die Sende-Funktionen sende_llc0_event..sende_llc3_event arbeiten eng verzahnt zusammen mit den Empfangs-Funktionen empfange_llc0_event..empfange_llc3_event. Sie realisieren Parallelität. Die Empfangs-Funktionen werden im Kapitel „Die LLC-(Logical Link Control) Teilschicht erklärt. Hier zunächst eine Aufstellung der Sende-Funktionen. Siehe Bilder 42, 43, 44, 45. Alle Funktionen überprüfen als erstes ihren llc_event_queue_kopf, ob bereits 1 Element in die Queue eingekettet ist.

- Wenn **nein** (**anzahl_der_llc_events** ist **0**), kettet die Funktion das llc_event-Element in die Queue ein, und überprüft anschließend, ob die Empfangs-Task im LLC Layer darauf wartet, daß die betreffende Queue nicht mehr leer ist (**if(condition_queue_kopf.....**)).
  Die Empfangs-Task wartet auf das Eintreffen des llc_event-Elements nur eine begrenzte Zeit. Sie hat daher einen Timer gestartet, der die Wartezeit begrenzt. Da das llc_event-Element eingetroffen ist, löscht die Sende-Funktion und damit die RxMTask diesen timer (timer_...._clear()) und signalisiert der Empfangs-Task, daß die Queue nicht mehr leer ist. Sie wird aus dem Wartezustand befreit (**signal(.....)**).
- Wenn **ja** (**anzahl_der_llc_events ist nicht 0**), ist die betreffende Queue voll. Die llc_event-Queues nehmen maximal 1 llc_event-Element auf. Die Sende-Funktion beeendet daraufhin ihre Arbeit, und das bedeutet, das eingetroffene llc_event-Element bzw. die eingetroffene LLC PDU wird „verworfen“. Man könnte zwar die RxMTask solange in den Wartezustand versetzen, bis die Queue nicht mehr voll ist, doch dann wäre die Receive Machine nicht mehr in der Lage, LLC PDUs von anderen Quell-Rechnern zu empfangen. Verworfene LLC PDUs gehen aber nicht verloren. Das Sicherungsprotokoll im LLC Layer sorgt für die wiederholte Übertragung und zwar solange, bis sie sicher angekommen sind.
  Siehe Kapitel „Die LLC-(Logical Link Control) Teilschicht. Noch ein Hinweis: Genaue Informationen zu den Mailbox-Mechanismen sind in Echtzeit-Multitasking Teil 2: Der Kernel zu finden.

```
#include<struct3.h>
#include<struct6.h>

#define llc0_event_queue_nicht_mehr_leer   31
#define llc_event_max                       4
#define condition_max                      50

extern llc_event_queue_kopf_struct  llc_event_queue_kopf[llc_event_max];
extern condition_queue_kopf_struct  condition_queue_kopf[condition_max];

void sende_llc0_event(llc_event_struct *llc_event_adr)
{
 llc_event_queue_kopf_struct    *llc_event_queue_kopf_adr;

 if(llc_event_queue_kopf[0].anzahl_der_llc_events == 0)
  {
   llc_event_queue_kopf_adr  = &llc_event_queue_kopf[0|;
   einketten(llc_event_queue_kopf_adr,llc_event_adr);

   if(condition_queue_kopf[llc0_event_queue_nicht_mehr_leer].
                                anzahl_der_condition_elemente !=0)
    {
     timer_task6_clear();
     signal(llc0_event_queue_nicht_mehr_leer);
    }
  }
}
```

Bild 42: Betriebssystem-Funktion sende_llc0_event (Datei sendllc0.c)

```
#include<struct3.h>
#include<struct6.h>

#define llc1_event_queue_nicht_mehr_leer    33
#define llc_event_max                        4
#define condition_max                       50

extern llc_event_queue_kopf_struct  llc_event_queue_kopf[llc_event_max];
extern condition_queue_kopf_struct  condition_queue_kopf[condition_max];

void sende_llc1_event(llc_event_struct *llc_event_adr)
{
 llc_event_queue_kopf_struct    *llc_event_queue_kopf_adr;

 if(llc_event_queue_kopf[1].anzahl_der_llc_events == 0)
  {
   llc_event_queue_kopf_adr  = &llc_event_queue_kopf[1|;
   einketten(llc_event_queue_kopf_adr,llc_event_adr);

   if(condition_queue_kopf[llc1_event_queue_nicht_mehr_leer].
                              anzahl_der_condition_elemente !=0)
    {
     timer_task7_clear();
     signal(llc1_event_queue_nicht_mehr_leer);
    }
  }
}
```

Bild 43: Betriebssystem-Funktion sende_llc1_event (Datei sendllc1.c)

```
#include<struct3.h>
#include<struct6.h>

#define llc2_event_queue_nicht_mehr_leer    35
#define llc_event_max                        4
#define condition_max                       50

extern llc_event_queue_kopf_struct  llc_event_queue_kopf[llc_event_max];
extern condition_queue_kopf_struct  condition_queue_kopf[condition_max];

void sende_llc2_event(llc_event_struct *llc_event_adr)
{
 llc_event_queue_kopf_struct    *llc_event_queue_kopf_adr;

 if(llc_event_queue_kopf[2].anzahl_der_llc_events == 0)
  {
   llc_event_queue_kopf_adr  = &llc_event_queue_kopf[2];
   einketten(llc_event_queue_kopf_adr,llc_event_adr);

   if(condition_queue_kopf[llc2_event_queue_nicht_mehr_leer].
                              anzahl_der_condition_elemente !=0)
    {
     timer_task8_clear();
     signal(llc2_event_queue_nicht_mehr_leer);
    }
  }
}
```

Bild 44: Betriebssystem-Funktion sende_llc2_event (Datei sendllc2.c)

```
#include<struct3.h>
#include<struct6.h>

#define llc3_event_queue_nicht_mehr_leer    37
#define llc_event_max                        4
#define condition_max                       50

extern llc_event_queue_kopf_struct  llc_event_queue_kopf[llc_event_max];
extern condition_queue_kopf_struct  condition_queue_kopf[condition_max];

void sende_llc3_event(llc_event_struct *llc_event_adr)
{
 llc_event_queue_kopf_struct    *llc_event_queue_kopf_adr;

 if(llc_event_queue_kopf[3].anzahl_der_llc_events == 0)
   {
    llc_event_queue_kopf_adr  = &llc_event_queue_kopf[3|;
    einketten(llc_event_queue_kopf_adr,llc_event_adr);

    if(condition_queue_kopf[llc3_event_queue_nicht_mehr_leer].
                                anzahl_der_condition_elemente !=0)
     {
      timer_task9_clear();
      signal(llc3_event_queue_nicht_mehr_leer);
     }
   }
}
```

Bild 45: Betriebssystem-Funktion sende_llc3_event (Datei sendllc3.c)

1.3.1.4.2 Botschaften an die Access Control Machine

Die Receive Machine schickt an die Access Control Machine drei verschiedene Botschaften:

- eine **ok**-Botschaft, wenn ein unbeschädigter **Daten**- oder **Response**-Frame angekommen ist,
- eine ok-Botschaft, wenn ein unbeschädigter **Token**-Frame angekommen ist oder
- eine **io_error**-Botschaft, wenn ein Frame beschädigt angekommen ist.

Zur Erinnerung: Die Receive Machine benutzt zum Transport der Botschaft den SAP (Service Access Point) **sende_acm_event(acm_event,rxm_puffer);**

Wie die Receive Machine, ist auch die Access Control Machine als Task implementiert (**ACMTask**); beide halten sich im MAC Layer auf. Da ihre Adreßräume voneinander isoliert sind, kann der Botschaften-Transport nur über eine Mailbox erfolgen. So unterhält die Access Control Machine eine eigene Mailbox, und die Receive Machine kettet dort die entsprechenden Informationen ein. Um welche Informationen handelt es sich?

Die Access Control Machine ist

- am Ereignis interessiert: acm_event = ok oder acm_event = io_error und
- am Control-Feld der MAC PDU: ns (next station), ts (this station), ctrl (control)

Siehe Bild 46:

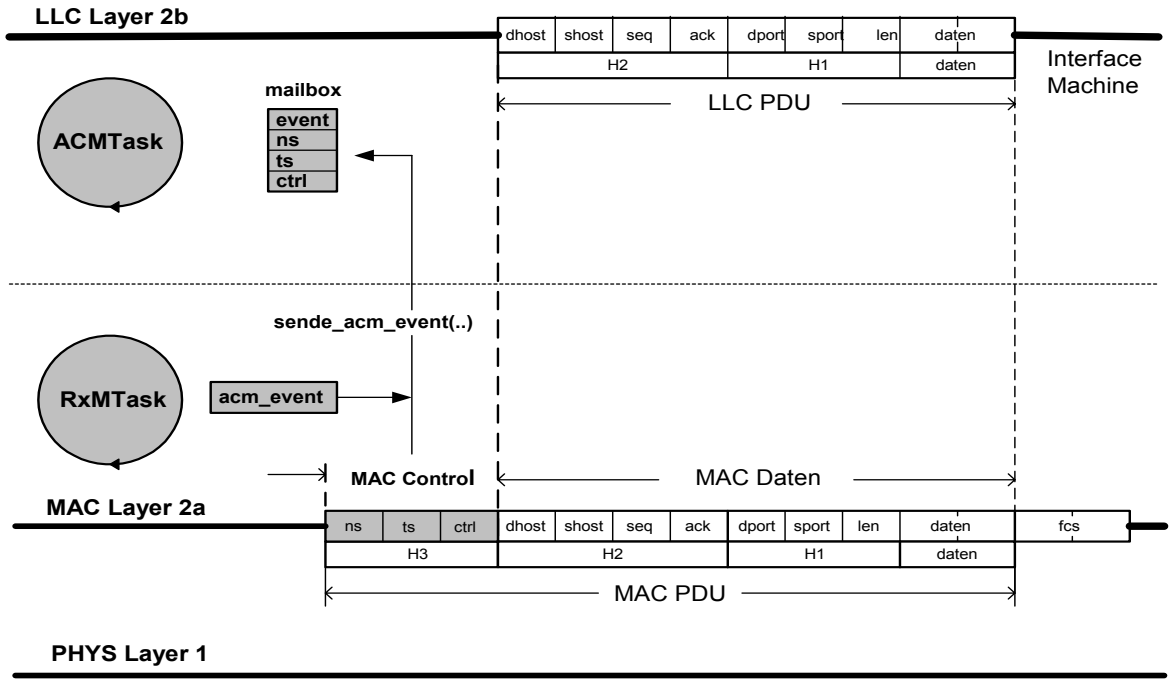


Bild 46: Botschaft an die Access Control Machine

Die Mailbox ist eine Queue und besteht aus einem **acm_event_queue_kopf** und einem **acm_event**-Element. Die Struktur dieser Queue ist durch die folgenden zwei Datentypen definiert:

```
typedef struct s19
        {
         struct s20       *erster_der_queue;
         struct s20       *letzter_der_queue;
         short int         anzahl_der_events;
        } acm_event_queue_kopf_struct;

typedef struct s20
        {
         struct s20       *naechster_in_der_queue;
         char              event;
         char              frame_control;
         unsigned char     dest_station;
         unsigned char     source_station;
        } acm_event_struct;
```

Bild 47: Definition der Datentypen acm_event_queue_kopf_struct und acm_event_struct

Die Datentypen heißen **s19** und **s20**. Beide sind umbenannt, und zwar s19 in acm_event_queue_kopf_struct und s20 in acm_event_struct.

- Eine Variable des Typs acm_event_queue_kopf_struct repräsentiert den Kopf der Queue und
- eine Variable des Typs acm_event_struct repräsentiert das **Ereignis** durch char event und das **MAC Control**-Feld durch char frame_control...unsigned char source_station.

Die beiden Zeiger erster_der_queue und letzter_der_queue im llc_event_queue_kopf zeigen auf das erste und letzte acm_event-Element in der Queue. Die short int-Variable anzahl_der_events zählt die Anzahl der eingeketteten acm_event-Elemente, und der Zeiger naechster_in_der_queue in einem acm_event-Element zeigt auf das nachfolgende Element. Damit das aufgetretene Ereignis zusammen mit den empfangenen MAC Control-Informationen in die Queue eingekettet werden kann, muß das Betriebssystem die Queue zur Verfügung stellen. So installiert unmittelbar nach dem Systemstart die Betriebssystem-Funktion init einen **acm_event_queue_kopf** und vier **acm_event**-Elemente. Siehe Bild 48.

```
#define null (void*)          0
#define acm_event_max     4
                                  •
                                  •
#include<struct9.h>
                                  •
                                  •

acm_event_queue_kopf_struct acm_event_queue_kopf;
acm_event_struct          acm_event[acm_event_max];
                                  •
                                  •
void init(void)
{
            /*  1 ACM-Event-Kopf installieren */

 acm_event_queue_kopf.erster_der_queue      = null;
 acm_event_queue_kopf.letzter_der_queue     = null;
 acm_event_queue_kopf.anzahl_der_events     = 0;

          /* 4 ACM-Event-Elemente installieren */

 for(i=0;i<acm_event_max;i++)
   {
    acm_event[i].naechster_in_der_queue  = null;
    acm_event[i].event                   = 0;
    acm_event[i].frame_control           = 0;
    acm_event[i].dest_station            = 0;
    acm_event[i].source_station          = 0;
   }
}
```

Bild 48: Fragment der Betriebssystem-Funktion init (Datei init.c)

Die „Kommunikationsstruktur“ zwischen der RxMTask und der ACMTask gestaltet sich danach wie folgt. Siehe Bild 49.

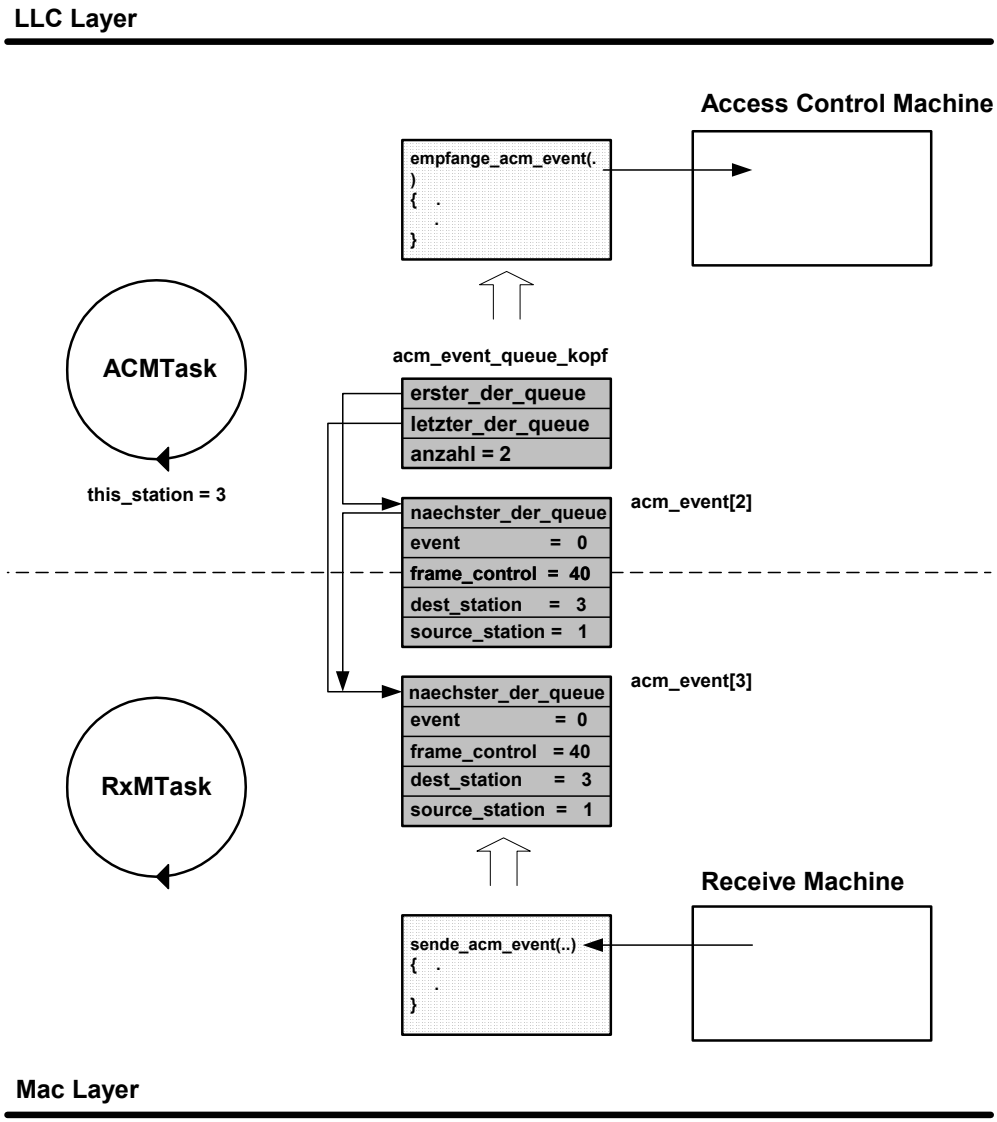


Bild 49: Kommunikation zwischen RxMTask und ACMTask

Das Bild demonstriert zusätzlich ein Beispiel. Die Station hat die Identifikations-Nummer 3 (this_station = 3). Diese Nummer darf nicht mit der host-Nummer verwechselt werden. Aus den eingeketteten acm_event-Elementen ist zu erkennen, daß die Quell-Station 1 (source_station = 1) an die Ziel-Station 3 (dest_station = 3) zwei Daten-Frames (frame_control = 0x40) geschickt hat.
Das Einketten besorgt die Receive Machine mittels der Funktion

- **sende_acm_event(char event, unsigned char rxm_puffer[]);**

Sie überträgt **event = 0** (ok) und die **MAC Control**-Informationen frame_control, dest_station und source_station. Diese Informationen entnimmt die Funktion den rxm_puffer-Elementen rxm_puffer[ns], rxm_puffer[ts] und rxm_puffer[ctrl].

Die Access Control Machine kann jetzt mittels der Funktion

- **empfange_acm_event(....);**

die eingeketteten acm_event-Elemente ausketten, die empfangenen Informationen auswerten, und ihre spezifischen Aufgaben erledigen. Die Sende-Funktion sende_acm_event arbeitet eng verzahnt zusammen mit der Empfangs-Funktion empfange_acm_event. Sie realisieren Parallelität. Die Empfangs-Funktion wird im Abschnitt „Die Access Control Machine“ erklärt. Hier zunächst die Sende-Funktion; siehe Bild 50.

```
#include<struct3.h>
#include<struct9.h>

#define acm_event_queue_nicht_mehr_leer    43
#define condition_max                      50
#define acm_event_max                       1
#define ns                                  0
#define ts                                  1
#define ctrl                                2
#define dhost                               3
#define shost                               4
#define not_identified                      0
#define data                             0x40
#define ok                                  0

extern char                              acm_event_in;
extern unsigned char                     this_station,host_id;
extern condition_queue_kopf_struct       condition_queue_kopf[condition_max];
extern acm_event_queue_kopf_struct       acm_event_queue_kopf;
extern acm_event_struct                  acm_event[acm_event_max];

void sende_acm_event(char event,unsigned char rxm_puffer[])
{
 acm_event_queue_kopf_struct   *acm_event_queue_kopf_adr;
 acm_event_struct              *neu_acm_event_adr;

 switch(event)
 {
  case ok: if((rxm_puffer[ctrl] & 0x40) == data)
           {
            if(rxm_puffer[dhost] == host_id)
              rxm_puffer[ns] = this_station;
            else
              rxm_puffer[ns] = not_identified;
           }
  break;
  default: break;
 }
```

```
if(acm_event_queue_kopf.anzahl_der_events < acm_event_max)
  {
   acm_event[acm_event_in].event              = event;
   acm_event[acm_event_in].dest_station       = rxm_puffer[ns];
   acm_event[acm_event_in].source_station     = rxm_puffer[ts];
   acm_event[acm_event_in].frame_control      = rxm_puffer[ctrl];
   acm_event[acm_event_in].source_host        = rxm_puffer[shost];

   acm_event_queue_kopf_adr  = &acm_event_queue_kopf;
   neu_acm_event_adr         = &acm_event[acm_event_in];

   acm_event_in++;
   if(acm_event_in == acm_event_max)
      acm_event_in =  0;

   einketten(acm_event_queue_kopf_adr,neu_acm_event_adr);

   if(condition_queue_kopf[acm_event_queue_nicht_mehr_leer].
                                          anzahl_der_condition_elemente !=0)
     {
      timer_task5_clear();
      signal(acm_event_queue_nicht_mehr_leer);
     }
  }
}
```

Bild 50: Betriebssystem-Funktion sende_acm_event (Datei sendacm.c)

Die Funktion überprüft als erstes den Parameter **event**. Ist event == **0** (ok), hat eine Quell-Station (source_station = ?) der betrachteten Station (dest_station = ?) entweder einen Token- oder einen Daten-Frame geschickt. Ist event != 0 ist ein beschädigter Frame angekommen. Nehmen wir als erstes an, es ist ein Token-Frame übertragen worden. Dazu demonstriert Bild 51 ein Beispiel.

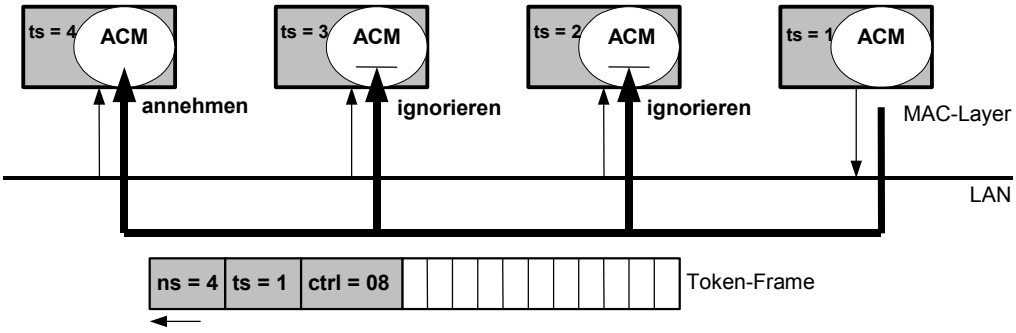


Bild 51: Station 1 schickt Station 4 einen Token-Frame

Die source_station 1 ist im Augenblick im Besitz des Token. Sie gibt das Token weiter an ihre Nachfolge-Station 4 bzw. dest_station 4, und schickt zu diesem Zweck einen Token-Frame (ctrl = 08) ins LAN. Die Stationen 2, 3 und 4 empfangen den Token-Frame, und die Receive Machine jeder Station gibt das MAC Control-Feld weiter an ihre Access Control Machine (ACM).

Die Access Control Machine der dest_station (ns=4) nimmt das MAC Control-Feld von Station 1 an, während die Access Control Machines der Stationen 2 und 3 das MAC Control-Feld ´ignorieren´. Siehe Abschnitt -Die Access Control Machine-.
Die Übergabe des MAC Control-Felds eines Daten-, Token- oder Error-Frames an die Access Control Machine geschieht folgendermaßen:
Die Funktion sende_acm_event überprüft ihren acm_event_queue_kopf, ob noch Elemente in der Queue Platz haben.

- Wenn **ja** (**anzahl_der_acm_events < acm_event_max**), überträgt die Funktion die rxm_puffer-Elemente rxm_puffer[ns], rxm_puffer[ts] und rxm_puffer[ctrl] in die korrespondierenden Felder des aktuellen acm_event[acm_event_in]-Elements, und kettet das betreffende Element acm_event[0] oder...acm_event[3] in die Queue ein.

  Hinweis: Zusätzlich wird noch die Nummer des source-host, nämlich rxm_puffer[shost] übertragen. Dies ist an dieser Stelle ungewöhnlich, doch die Maßnahme erlaubt der ACM-Logik die Behandlung einer besonderen Situation, wie wir im Abschnitt -Die Access Control Machine- erkennen werden. struct9.h ist aus diesem Grund um die zusätzliche Variable source_host erweitert.

  Anschließend überprüft die Funktion, ob die ACMTask darauf wartet, daß die betreffende Queue nicht mehr leer ist (if(condition_queue_kopf.....)).
  Die ACMTask wartet auf das Eintreffen des acm_event-Elements nur eine begrenzte Zeit. Sie hat daher einen Timer gestartet, der die Wartezeit begrenzt. Da ein acm_event-Element eingetroffen ist, löscht die Sende-Funktion und damit die RxMTask diesen timer (timer_taskx_clear()) und signalisiert der ACM-Task, daß die Queue nicht mehr leer ist. Sie wird aus dem Wartezustand befreit (**signal**(.....)).

- Wenn **nein** (**anzahl_der_acm_events == acm_event_max**), „verwirft“ die Sende-Funktion den eingetroffenen Frame.

Was geschieht, wenn die Sende-Funktion **(rxm_puffer[ctrl] & 0x40) == data** feststellt? Das ctrl-Feld enthält entweder den Wert 40H oder 50H. Eine Quell-Station hat der betrachteten Ziel-Station einen Daten- oder Response-Frame geschickt. An einem solchen Frame ist zwar die Access Control Machine interessiert, sie bekommt aber keinerlei Informationen darüber, ob der betreffende Frame für ihre Station bestimmt ist. So muß die Sende-Funktion sende_acm_event Hilfestellung leisten. Sie hat Zugriff auf alle rxm_puffer-Elemente, und so überprüft sie mit

if(rxm_puffer[dhost] == host_id),

ob der übertragene Daten-Frame den betrachteten host adressiert. Wenn ja, trägt die Sende-Funktion die Stations-Nummer **this_station** in das Element rxm_puffer[ns] ein. Wenn nein, bekommt dieses Feld den Wert 0 (not_identified).
Ist **ns != 0**, startet die Access Control Machine besondere Aktionen.
Ist **ns == 0**, wird die Access Control Machine lediglich darüber informiert, daß sich ein Daten-Frame im LAN befunden hat. Dies ist auch der Grund, warum in einem tokenbus-basierten LAN die Stations-Nummer 0 (this_station = 0) nicht vorkommt.
Betrachten wir hierzu folgendes Szenarium. Siehe Bild 52.

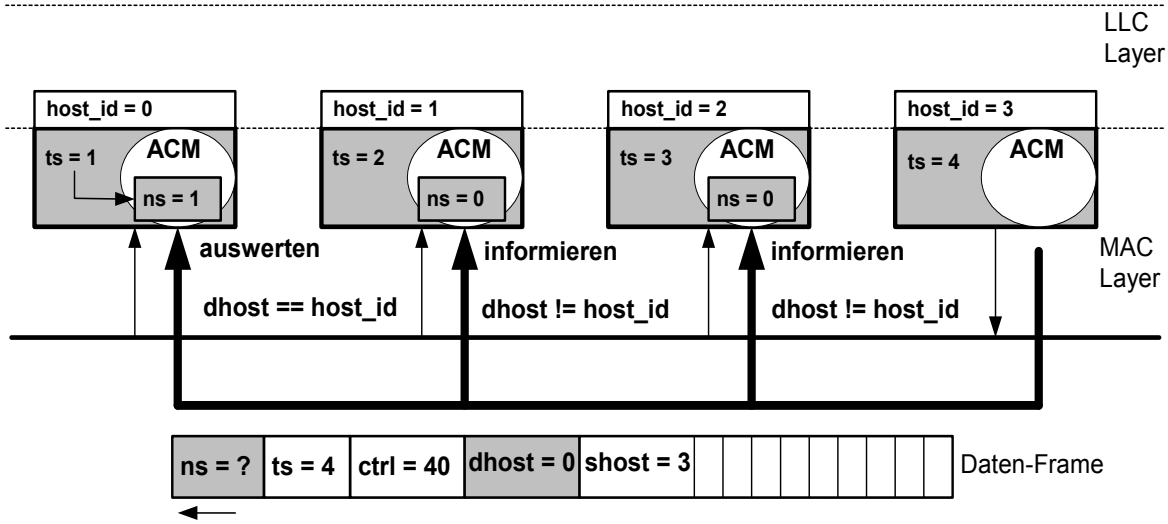


Bild 52: Behandlung eines Daten-Frame

Die LLC Layer haben ihren Rechnern die Identifikations-Nummern host_id = 0...host_id = 3 gegeben, und die MAC Layer die Identifikations-Nummern ts =1...ts = 4. Der Quell-Rechner 3 (host_id = 3 bzw. **shost = 3**) schickt dem Ziel-Rechner 0 (**host_id = 0** bzw. dhost = 0) einen Daten-Frame (**ctrl = 40**). Der Frame kommt bei den hosts 0, 1 und 2 an (host_id = 0, 1, 2) bzw. bei den Stationen 1, 2 und 3 (ts = 1, 2, 3). Die Sende-Funktionen sende_acm_event der Stationen ts=2 und ts=3 erkennen, daß **dhost != host_id** ist. So initialisieren die Sende-Funktionen das ns-Feld mit 0 (**ns = 0**) und geben die resultierenden MAC Control-Informationen weiter an ihre Access Control Machines. Sie werden auf diese Weise über den Daten-Frame-Transfer im LAN informiert.
Die Sende-Funktion der Station 0 erkennt, daß **dhost == host_id** ist. Sie initialisiert das ns-Feld mit ts = 1 (**ns = ts =1**), und gibt auch hier die resultierenden MAC Control-Informationen weiter an ihre Access Control Machine. Diese wertet die Informationen aus und startet die bereits angedeuteten speziellen Aktionen.
Die „Die Receive Machine“ ist nun in allen seinen Funktionen beschrieben, und so können wir uns der nächsten logischen Einheit des MAC Layers, nämlich der Access Control Machine, zuwenden.

1.3.2 Die Access Control Machine

Die Access Control Machine hält sich wie alle anderen MAC-Machines in der Privileg-Ebene 0 auf. Sie ist zwar eine System-Task, wird aber vom Scheduler wie eine ´normale´ Anwender-Task behandelt. Das heißt, sie unterliegt dem Round Robin- und MLF (Multilevel Feedback)-Scheduling und ihre Priorität ist 2. Identifiziert ist sie durch die folgenden beiden Datenstrukturen:

- Task-Status-Segment (TSS) **ACM_Task** und
- Task-Control-Block (TCB) **task_cb[5]**.

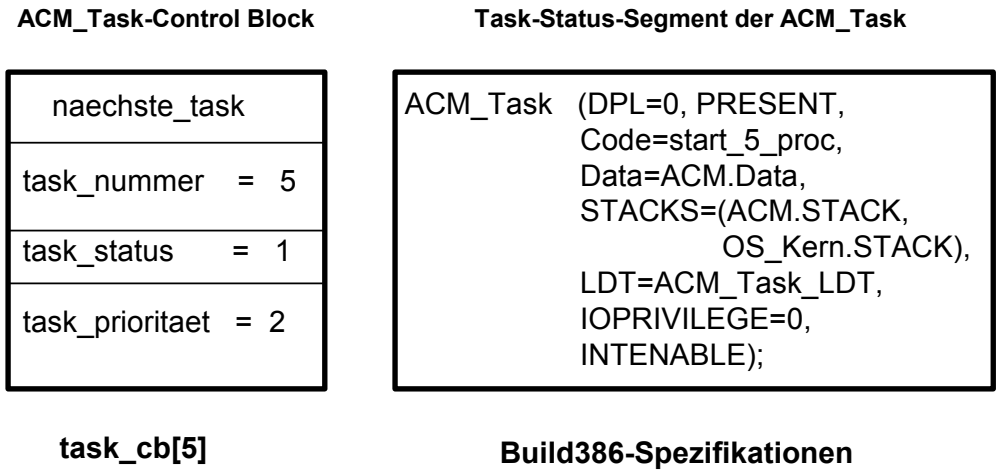


Bild 53: Identifikation der ACM_Task

Auf die Build386-Spezifikationen und die Initialisierung des Task-Control-Blocks soll an dieser Stelle nicht eingegangen werden.

Bevor die ACM_Task die Access Control Machine startet, ruft sie die zugänglichen Funktion **acm_init(.)** auf, die von der Privileg-Ebene 3 aus über den Call Gate-Deskriptor **KN_ACM_init** und dem Assembler-Adapter **adapt017** erreichbar ist. Diese Funktion initialisiert einige Variablen, die für die Access Control Machine von Bedeutung sind. Innerhalb einer Endlosschleife erfolgt anschließend der Aufruf der zugänglichen Funktion **acm_exe()**. Sie realisiert die Funktionen der Access Control Machine und ist über den Assembler-Adapter **adapt018** erreichbar. Die ACM_Task kehrt innerhalb der while(1)-Schleife immer wieder zur Privileg-Ebene 3 zurück. Sie ist nur dort zur Realisierung des Round Robin- und MLF-Scheduling von den Clock-Ticks unterbrechbar. Siehe Bild 54.

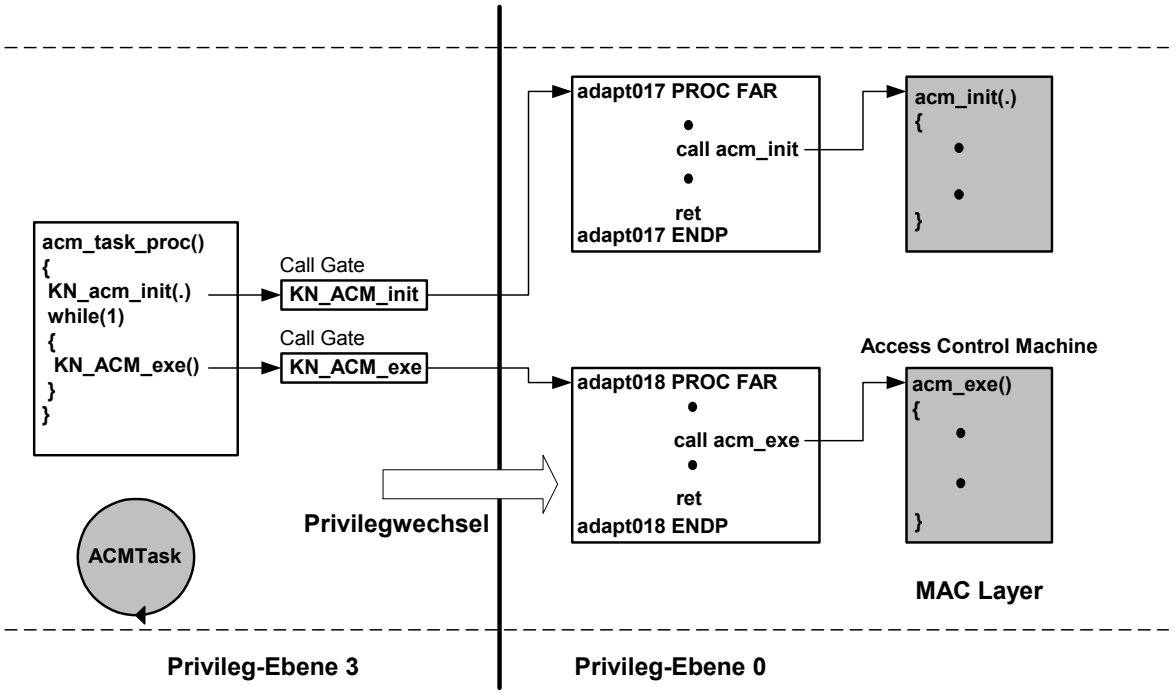


Bild 54: Initialisierung und Aufruf der Access Control Machine durch die ACM_Task

Im Initialisierungsaufruf übergibt die ACM_Task an die Betriebssystem-Funktion acm_init(.) einen Parameter, der die sogenannte Token-Haltezeit bzw. **token hold time** enthält. Wir erinnern uns: Ist eine Station im Besitz des Token, ist sie nur eine begrenzte Zeit sendeberechtigt. Während dieser Zeit hält die Station das Token, und gibt das Token nach Ablauf der Zeit weiter an die Nachfolgestation.
Nach IEEE 402.4 sind für die Sende-Daten optional 4 verschiedene Prioritätsstufen, sogenannte **„access classes“** vorgegesehen. Sie repräsentieren die Dringlichkeit der zu übertragenden Daten. Sie haben die Bezeichnungen 0, 2, 4 oder 6. Dabei hat 6 die höchste Priorität und 0 die niedrigste. In diesem Text betrachten wir nur die access class mit der höchsten Priorität. Die token hold time an dieser Stufe heißt **hi_pri_token_hold_time**. Das folgende Bild 55 zeigt die beiden Funktions-Aufrufe der ACM_Task in der Privileg-Ebene 3.

```
extern void far KN_ACM_init(unsigned short int);
extern void far KN_ACM_exe(void);

static unsigned short int hi_pri_token_hold_time;

acm_task_proc()
{
 hi_pri_token_hold_time=100;      /* Token-Haltezeit = 100ms */
 KN_ACM_init(hi_pri_token_hold_time);

 while(1)
 {
  KN_ACM_exe();
 }
}
```

Bild 55: System-Modul ACM_Task (Datei acmtask.c)

1.3.2.1 Access Control Machine (ACM)-Zustände und Transitionen

Die ACM-Logik einer Station kann als eine Bearbeitungs-Maschine beschrieben werden, die eine Anzahl von unterschiedlichen Phasen durchläuft. Diese Phasen bezeichnen wir als Zustände bzw. **States**, und die Übergänge als **Transitionen**. Somit gestaltet sich die ACM-Logik als eine endliche Zustandsmaschine bzw. **finite state machine** (fsm), die von der Betriebssystem-Funktion acm_exe() realisiert wird. Nach IEEE 802.4 sind insgesamt 11 Zustände definiert, doch wir werden uns auf die 6 wichtigsten beschränken. Jeder Zustand ist durch eine individuelle Nummer identifiziert. Diese Zustände und ihre Identifikationsnummern lauten wie folgt:

Idle	= 1
Claim_token	= 4
Use_Token	= 5
Await_Ifm_Response	= 6
Pass_Token	= 8
Check_Pass_Token	= 9

Die ACM arbeitet eng zusammen mit der Receive Machine (RxM), der Sende-Einheit der Interface-Machine (IFM) und der Transmit Machine (TxM). Wie das Korrespondenz-Schema, abhängig von den Zuständen der ACM aussieht, zeigt das folgende Bild 56.

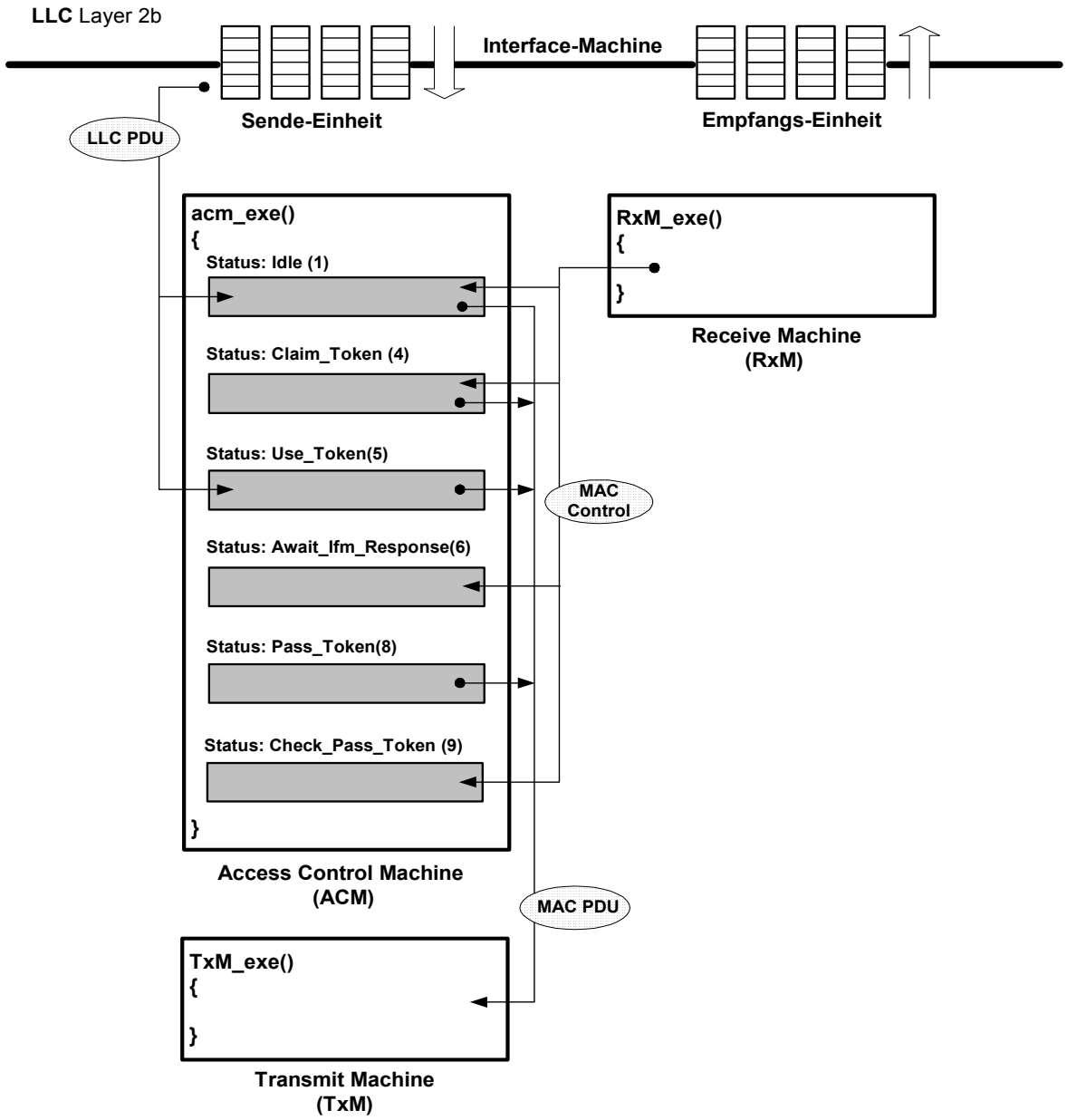


Bild 56: Zustände und Korrespondenzen der Access Control Machine

Wie zu sehen ist, **empfängt** die ACM

- in den Zuständen Idle (1), Claim_Token(4), Await_Ifm_Response (6) und Check_Pass_Token (9) **MAC Control**-Informationen von der Receive Machine und
- in den Zuständen Idle (1) und Use-Token(5) **LLC PDU**s von der Sende-Einheit der Interface Machine

Sie **sendet**

- in den Zuständen Idle(1), Claim_Token(4), Use_Token (5) und Pass_Token (8) **MAC PDU**s an die Transmit Machine.

Die Betriebssystem-Funktion acm_exe() benutzt eine Anzahl von Variablen, die während der Initialisierungs-Phase von acm_init(.) definiert und initialisiert werden. Siehe Bild 57.

```
#define idle                    1
#define mac_pdu_max            14
#define max_access_class        6

unsigned char           mac_pdu[mac_pdu_max] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned char           send_pending[max_access_class+1] = {0,0,0,0,0,0,0};
unsigned char           llc_pdu_pending;
unsigned short int      hi_pri_token_hold_time;
char                    frame_control;
char                    acm_status;

void acm_init(unsigned short int hptht)
{
 hi_pri_token_hold_time = hptht;
 acm_status             = idle;
}
```

Bild 57: Betriebssystem-Funktion acm_init (Datei acm_init.c)

Im Laufe des Textes werden wir auf die Bedeutung der einzelnen Variablen eingehen. Bekannt ist uns bereits der Parameter hptht. Er enthält die vom Anwender in der Privileg-Ebene 3 gewählte Token-Haltezeit der access class 6, und acm_init(.) initialisiert damit die Privileg-Ebene 0-Variable **hi_pri_token_hold_time**.
Von primären Interesse ist zunächst die Variable **acm_status**; acm_init(.) initialisiert sie mit 1 und bringt damit die Access Control Machine in den Anfangszustand **idle**. In diesem Zustand startet anschließend die Betriebssystem-Funktion acm_exe(). Sie überprüft im Laufe ihrer Arbeit eine Menge von Ereignissen, die größtenteils von der Receive Machine gemeldet werden, und bringt davon abhängig, die Access Control Machine in die eingangs erwähnten Zustände.

Wie die Funktion acm_exe() strukturiert ist, zeigt das folgende Bild 58.

```
#define idle                    1
#define claim_token             4
#define use_token               5
#define await_ifm_response      6
#define pass_token              8
#define check_pass_token        9

extern char    acm_status;

void acm_exe()
{
 switch(acm_status)
 {
  case idle:
      acm_status = use_token;
      acm_status = claim_token;
  break;

  case claim_token:
      acm_status = idle;
      acm_status = use_token;
  break;

  case use_token:
      acm_status = await_ifm_response;
      acm_status = pass_token;
  break;

  case await_ifm_response:
      acm_status = use_token;
      acm_status = idle;
      acm_status = use_token;
  break;

  case pass_token:
      acm_status = check_pass_token;
  break;

  case check_pass_token:
      acm_status = use_token;
      acm_status = idle;
  default: break;
 }
}
```

Bild 58: Struktur der Betriebssystem-Funktion acm_exe()

In jedem Zustand werden eine Anzahl von Ereignissen überprüft, deren Auswertungen zu neuen Zuständen führen. Die Übergänge (Transitionen) realisiert die Funktion mit Hilfe der Anweisung **acm_status** = ***neuer_zustand***. So können z.B. im Zustand **idle** Ereignisse auftreten, die den künftigen Zustand **use_token** oder **claim_token** zur Folge haben.

Hat acm_exe() die Variable acm_status mit dem künftigen Zustand der Access Control Machine initialisiert, veranlasst switch(acm_status) die Fortsetzung des Programms im neuen Zustand. So lässt sich die ACM-Logik als **finite state machine** darstellen, die im folgenden Bild 59 illustriert ist.

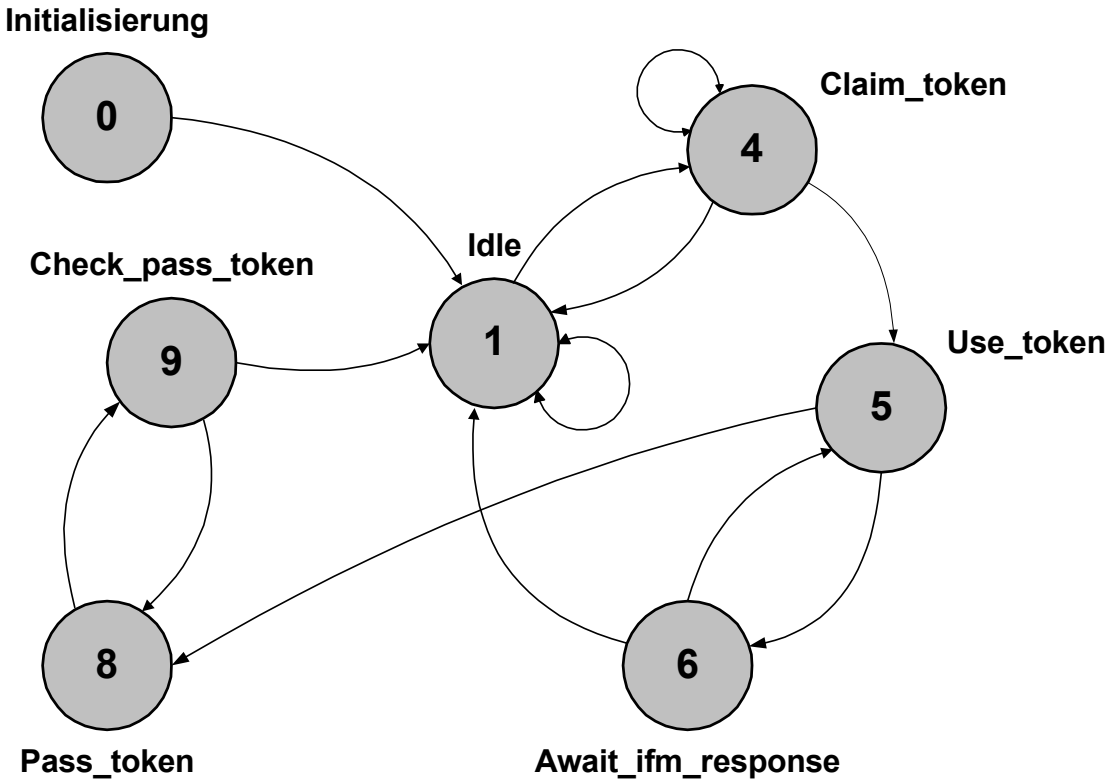


Bild 59: ACM Finite State Machine-Diagramm

Wir wollen nun die einzelnen Zustände und die darin enthaltenen Aktionen betrachten. Die Betriebssystem-Funktion acm_exe() benutzt dabei eine Anzahl von operationalen Variablen und Definitionen, die teilweise extern oder im acm_exe-Modul selbst definiert sind. Im Laufe des Textes werden wir darauf eingehen. Hier zunächst das acm_exe()-Programm-Fragment mit der Auflistung aller Definitionen und Variablen. Siehe Bild 60.

```
#define idle                        1
#define claim_token                 4
#define use_token                   5
#define await_ifm_response          6
#define pass_token                  8
#define check_pass_token            9
#define mac_pdu_max                14
#define claim                       0
#define token                    0x08
#define data                     0x40
#define response                 0x50
#define timeout                    -2
#define io_error                   -1
#define pass_count_max              5
#define station_max                 5
#define slot_time                  50
#define idle_time                   7*slot_time
#define ok                          0
#define ns                          0
#define ts                          1
#define ctrl                        2
#define max_access_class            6
#define enable                      1
#define disable                     0

extern unsigned char        mac_pdu[mac_pdu_max];
extern unsigned char        send_pending[max_access_class+1];
extern unsigned char        llc_pdu_pending;
extern unsigned char        first_station;
extern unsigned char        this_station;
extern unsigned char        next_station;
extern unsigned char        next_init;
extern char                 acm_status;
extern char                 frame_control;
extern char                 token_hold_timer;
extern unsigned short int   token_hold_timer;
extern unsigned short int   hi_pri_token_hold_time;

static char                 acm_event;
static unsigned char        dest_station;
static unsigned char        source_station;
static unsigned char        next_station;
static unsigned char        source_host;
static unsigned char        claim_pass_count;
static unsigned short int   bus_idle_timer;
static unsigned short int   token_pass_timer;
static unsigned short int   response_window_timer;
static unsigned char        L[pass_count_max][station_max];
```

Bild 60: Definitionen und Variablen der Betriebssystem-Funktion acm_exe()

1.3.2.1.1 Der Zustand Idle

In der Initialisierungs-Phase (Zustand 0) bringt die ACM_Task die Access Control Machine in den Zustand Idle (Zustand 1); siehe Bild 57. Danach ruft sie acm_exe() auf und beginnt mit der Ausführung der „Idle-Anweisungen“. Wie dieses Programm-Fragment aussieht, zeigt das folgende Bild 61.

```
void acm_exe()
{
 switch(acm_status)
 {
 case idle:
   clear_thtr();
   claim_pass_count = 1;
   uart_in_empfangs_modus();

   bus_idle_timer=idle_time;
   empfange_acm_event(bus_idle_timer,&acm_event,&frame_control,
                      &dest_station,&source_station,&source_host);
   switch(acm_event)
   {
    case ok:
       switch(frame_control)
       {
        case token:
           if(dest_station==this_station)
             {                                          /*    receive_token     */
              set_thtr(hi_pri_token_hold_time);
              acm_status = use_token;
             }
         break;
         case data:                                     /* transmit_response */
            if(dest_station == this_station)
             {
              ifm_response(source_host,&llc_pdu_pending,mac_pdu);
              if(llc_pdu_pending)
               {
                 mac_pdu[ns]   = source_station;
                 mac_pdu[ts]   = this_station;
                 mac_pdu[ctrl] = response;
                 uart_in_sende_modus();
                 txm_exe(mac_pdu);
               }
             }
         break;
       default: break;
       }
    break;

    case timeout:
      acm_status = claim_token;
    break;

    default: break;                                 /*          io_error        */
   }
  break;
 }
                          • •
}
```

Bild 61: acm_exe()-Programm-Fragment: ACM-Zustand Idle

Als erstes löscht die ACM mit **clear_thtr()**; den **token hold timer** und gibt danach der Variablen **claim_pass_count** den anfänglichen Wert 1. Diese Variable benötigt die ACM im Zustand claim_token. Welchen Dienst sie dort leistet, sehen wir etwas später. Schließlich bringt die ACM den UART in den Empfangs-Modus und ruft zu diesem Zweck die Betriebssystem-Funktion **uart_in_empfangs_modus()** auf. Siehe Bild 62.

```
#include<i86.h>

#define ir9_enable      0xfd
#define ier1            0x381
#define mcr1            0x384

extern unsigned short int  ocw1_reg_slave;
extern unsigned char       ocw1_slave;

uart_in_empfangs_modus()
{
 outbyte(mcr1,8);                /*  UART in Empfangs-Modus  */
 outbyte(ier1,0x05);             /*    FIFO-Interrupt zulassen    */
 ocw1_slave &= ir9_enable;
 outbyte(ocw1_reg_slave,ocw1_slave);  /*    IR9 zulassen      */
}
```

Bild 62: Betriebssystem-Funktion uart_in_empfangs_modus (Datei empfmod.c)

Mit **outbyte(mcr1,8)**; wird das **M**odem-**C**ontrol **R**egister des UART (seine Adresse ist 0x384) mit dem Wert 8 geladen. Damit wird sein RxD-Eingang (Receive Data) freigegeben und sein TxD-Ausgang (Transmit Data) gesperrt. Der UART ist im **Empfangs**-Modus. Vergleiche mit Bild 5: Halbduplex-Kommunikation mit RS485.
Mit **outbyte(ier1,0x05)**; wird das **I**nterrupt **E**nable **R**egister des UART (seine Adresse ist 0x381) mit dem Wert 5 geladen. Damit wird sein RxRDY-Ausgang (Receiver Ready) in die Lage versetzt, einen Low-Pegel und damit eine **Interrupt-Anforderung** zu generieren. Dies geschieht entweder beim Erreichen des Interrupt Trigger Levels (zur Erinnerung: 14 Oktets) oder durch einen Character Timout. Vergleiche mit Bild 13: Interrupt-Anforderung des UART. Gleichzeitig wird erlaubt, daß potenzielle Übertragungsfehler wie Overrun-, Parity- oder Frame-Error in das sogenannte Line Status Register (auch als Zustands-Register bekannt) eingetragen werden.
Wenn der UART eine Interrupt-Anforderung stellt, dann geschieht dies über den **IR9**-Eingang der Interrupt-Kaskade (siehe Bild 13). Mit ir9_enable = 0xfd wird das entsprechende Masken-Bit im OCW1-Register (**O**peration **C**ode **W**ord) des Interrupt-Controllers 1 (Slave) gelöscht und damit der Interrupt zugelassen. Realisiert wird dies mit **outbyte(ocw1_reg_slave,ocw1_slave)**; Dabei ist ocw1_reg_slave die Adresse des OCW1-Registers, nämlich **0xa1** (extern in der Betriebssystem-Funktion ioinit() definiert) und ocw1_slave ist der Initialisierungswert.

Fazit: Alle Stationen befinden sich zu Beginn der Idle-Phase im Empfangs-Modus und keine Station hat das Token. So lautet die Frage: Auf welche Weise bekommt eine Station das Token und damit das Recht zu senden?

> Eine einfache Idee ist folgende: Jede Station wird eine individuelle Zeitspanne in den Wartezustand versetzt. Diejenige Station, deren Wartezeit als erstes abgelaufen ist, nimmt sich das Token und sendet.

Wie können die individuellen Wartezeiten bestimmt werden? Wir wählen für alle Stationen die gleiche Ruhezeit bzw. idle_time und multiplizieren sie mit der Stationsnummer ts bzw. this_station. Es ergibt sich **bus_idle_timer=this_station • idle_time**.

Es sei idle_time = 50ms. Damit ergeben sich folgende individuelle Wartezeiten.

idle_time	this_station	this_station • idle_time (Wartezeit)
50ms	1	50ms • 1 = **50ms**
50ms	2	50ms • 2 = **100ms**
50ms	3	50ms • 3 = **150ms**
usw.		

Wir erkennen, dass diese Methode ein **Prioritätsschema** vorgibt. Die Station mit der niedrigsten Identifikationsnummer hat die höchste Priorität, und die Station mit der höchsten Identifikationsnummer die niedrigste. So dominiert Station 1 alle anderen Stationen und man könnte ihr von Anfang an das Token zusprechen. Doch in der Praxis können nicht alle Stationen gleichzeitig zugeschaltet werden und manche (inklusive Station 1) bleiben vielleicht für längere Zeit außerhalb des logischen Rings. Dann können auch Stationen mit höherer Nummer das Token gewinnen.
Diese Methode funktioniert zwar, ist aber viel zu schwerfällig. Man stelle sich vor, im Augenblick sind nur die Stationen mit den Nummern 250_{10}, 251_{10}, 252_{10} und 253_{10} im Ring. Dann wartet die Station 250_{10} die Zeit bus_idle_timer=250 • 50ms=12500ms (**12,5** s), bis sie endlich das Token in Besitz nehmen kann. Eine Wartezeit dieser Länge ist natürlich nicht akzeptabel und so suchen wir nach einer besseren Lösung.
Sie soll die maximale Wartezeit der Stationen auf ein Minimum reduzieren und sie soll für alle Stationen gleich sein.
Eine solche Lösung gibt es. Es handelt sich um den **claim token-**Algorithmus, der eine Modifikation des **binären Countdown**-Protokolls ist. Dieses Protokoll realisiert ein Verfahren, das anders als das token passing-Verfahren, den kollisionsfreien Zugriff auf das LAN regelt.
Wenn mehrere Stationen gleichzeitig ihre Frames übertragen möchten, kommen sie zwangsläufig in eine Konkurrenzsitution. Denn jede der beteiligten Stationen beansprucht das LAN für sich. Damit ein „Streit“ um das LAN gar nicht erst aufkommt, schickt **jede** Station vor der Frame-Übertragung ihre Adresse **ts** (this station) als binären Bitstring an **alle** anderen Stationen (broadcast) und beginnt dabei mit dem höchstwertigen Bit, dem MSB:

$$x = ts_1 \vee ts_2 \vee ..... \vee ts_N$$

Das Übertragungsmedium muss die Eigenschaft haben, dass ein Bit mit dem Wert 1 ein Bit mit dem Wert 0 überschreibt. Oder kurz, die „1“ muss die „0“ dominieren. Wenn eine Station erkennt, dass in ihrer Adresse eine höherwertige Bitposition 0 ist und an gleicher Stelle in x eine 1 steht, dann zieht sie ihre Bewerbung zurück.
Diese Art der Arbitrierung (Schiedsrichter-Algorithmus) ist im Bild 63 illustriert. Dort bewerben sich drei Stationen mit den Identifikationsnummern 48_{16}, 47_{16} und $2A_{16}$ simultan um das LAN.

Die MSBs sind in allen drei Stationen identisch „0“, so dass es hier zu keiner Dominanz kommt. Die Bits 6 der Stationen 48_{16} und 47_{16} sind „1“ und dominieren an dieser Stelle die „0“ der Station $2A_{16}$, die daraufhin aufgibt. An den Bitstellen 5 und 4 gibt es keine Dominanz. Doch an der Bitstelle 3 dominiert die „1“ von Station 48_{16} die „0“ von Station 47_{16}, die deshalb ebenfalls aufgibt. So bleibt als Gewinner die Station 48_{16}, die daraufhin ihre Frame-Übertragung beginnen kann. Weil in den übertragenen Stationsadressen die höherwertigen „1“-Signale dominieren, wird dieser Algorithmus auch als **Bit-Dominanz-Protokoll** bezeichnet. Man könnte es auch Bully-Protokoll nennen, frei nach dem Motto: Der stärkste Kerl in der Straße gewinnt. Das Prinzip dieses Protokolls dient uns als Grundlage für die Entwicklung des claim token-Algorithmus.

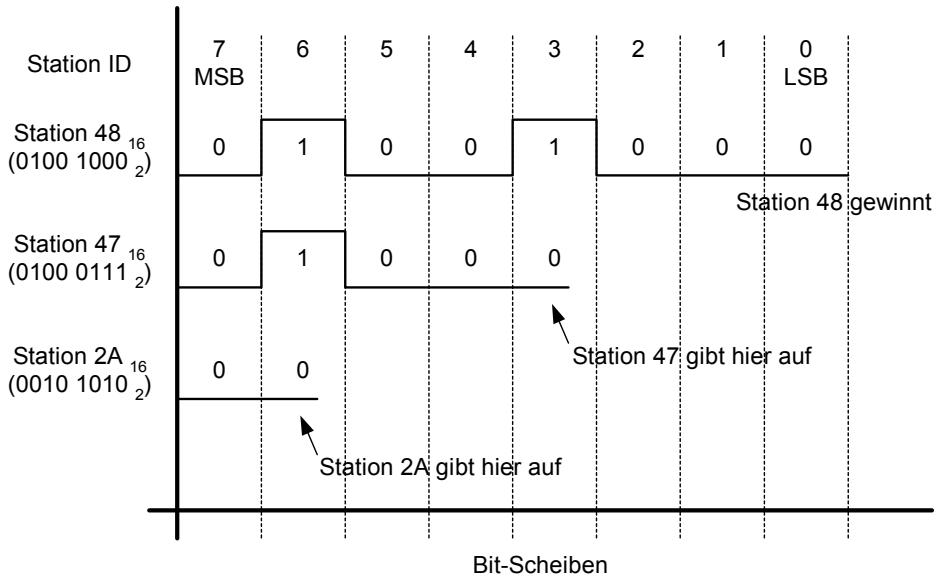


Bild 63: Arbitrierung im binären Countdown-Protokoll

Wenden wir uns nun der Empfangs-Funktion zu. Die Access Control Machine ruft sie mit

- **empfange_acm_event(bus_idle_timer,&acm_event,&frame_control, &dest_station,&source_station,&source_host);**

auf (siehe Bild 61) und empfängt damit die MAC Control-Informationen der Receive Machine. Sie ist somit das „Gegenstück“ zur Sende-Funktion sende_acm_event(..), die von der Receive Machine benutzt wird (vergleiche mit Bild 49: Kommunikation zwischen RxMTask und ACMTask). Wie zu sehen ist, wird die Funktion mit 5 Parametern aufgerufen:

- Der statischen Variablen **bus_idle_timer**.
  Sie wird vorher wegen #define idle_time **7*slot_time** und bus_idle_timer=idle_time mit der Dauer von 7 Zeitschlitzen initialisiert. Dieser Zahlenwert ist **nicht** willkürlich. Er wird verständlich, wenn wir uns später den claim_token-Algorithmus angesehen haben.
- Der Adresse der externen Variablen **frame_control**,
- den Adressen der statischen Variablen **acm_event**, **dest_station**, **source_station** und der bereits erwähnten
- besonderen Variablen **source_host**.

Alle diese Variablen sind im acm_exe()-Modul bzw. im acm_init()-Modul definiert (siehe Bilder 57 und 60). Die Implementierung der Empfangs-Funktion zeigt das folgende Bild 64:

```
#include<struct9.h>

#define acm_event_queue_nicht_mehr_leer    43

extern acm_event_queue_kopf_struct acm_event_queue_kopf;

void empfange_acm_event(unsigned short int zeit,char *event,
                                char               *frame_control,
                                unsigned char   *dest_station,
                                unsigned char   *source_station,
                                unsigned char   *source_host)
{
 acm_event_queue_kopf_struct    *acm_event_queue_kopf_adr;
 acm_event_struct                    *acm_event_adr;

 if(acm_event_queue_kopf.anzahl_der_events == 0)
   {
    timer_task5_start(zeit);
    wait(acm_event_queue_nicht_mehr_leer);
   }
                    /*  acmeventempfangen */

 acm_event_queue_kopf_adr   = &acm_event_queue_kopf;
 acm_event_adr                    = acm_event_queue_kopf_adr -> erster_der_queue;

 *event               = acm_event_adr -> event;
 *frame_control   = acm_event_adr -> frame_control;
 *dest_station      = acm_event_adr -> dest_station;
 *source_station   = acm_event_adr -> source_station;
 *source_host      = acm_event_adr -> source_host;

 ausketten(&acm_event_queue_kopf);
}
```

Bild 64: Betriebssystem-Funktion empfange_acm_event (Datei empfacm.c)

Die Funktion stellt als erstes fest, ob die acm_event_queue leer ist oder nicht. Wenn ja (anzahl_der_events == 0), startet sie die Zeitgeber-Funktion timer_task5_start(zeit); und „versorgt“ sie mit der Wartezeit. Danach führt sie ein **wait** aus, und die ACM_Task wartet, bis die **acm_event_queue_nicht_mehr_leer** ist. Aus diesem Zustand wird sie von zwei verschiedenen Quellen „befreit“. Entweder

1, von der Receive Machine. Sie meldet die erfogreiche Ankunft einer MAC Control-Information mit **event=ok** bzw. einen Übertragungsfehler mit **event=io_error** oder
2, von der Zeitgeber-Funktion. Sie meldet das Ende der Wartezeit mit **event=timeout**.

Beide Absender benutzen die Sende-Funktion sende_acm_event(.); wobei die Receive Machine bei einem Übertragungsfehler nur den Eintrag **acm_event[acm_event_in].event** benutzt. Die verbleibenden Felder bleiben verständlicherweise unberücksichtigt. Das gleiche gilt für die Zeitgeber-Funktion.

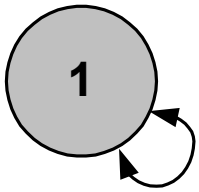
Wenn die acm_event_queue nicht leer war oder nach der Wartezeit nicht mehr leer ist, überträgt empfange_acm_event(.), die empfangenen Informationen in die übergebenen Parameter event, frame_control, dest_station, source_station und source_host. Anschließend wird das acm_event-Element aus der Queue ausgekettet. Danach setzt die Funktion acm_exe(.); im ACM Idle-Zustand ihre Arbeit fort. Sie überprüft als erstes das empfangene Ereignis acm_event.

```
switch( acm_event)
{
 case ok:
      •
      •
 break;
 case timeout:
      •
      •
 break;
 default break;          /*  io_error  */
 }
```

Sie stellt fest, ob die Übertragung ok war, ein timeout oder ein io_error aufgetreten ist (vergleiche mit Bild 61).

1, io_error

Bei dieser Botschaft ist entweder ein beschädigter Daten-Frame, ein beschädigter Token-Frame oder ein beschädigter Claim-Frame angekommen. So gibt die Access Control Machine dem Sender eine weitere Chance und bleibt im Idle-Zustand.

Anweisung	Transition
	Idle
default break;	

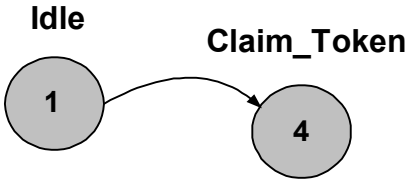
2, timeout

Die Zeitgeber-Funktion hat der Access Control Machine das Ende der **Bus-Ruhezeit** (bus_idle_timer=7*slot_time) signalisiert. Die Station bekommt deshalb die Berechtigung, sich um den Erhalt des Token zu bewerben. Zu diesem Zweck geht die ACM in den Zustand claim_token.

Anweisungen

```
case timeout:
      acm_status  =  claim_token;
break;
```

Transition



3, ok

Bei dieser Botschaft ist entweder ein Daten-Frame, ein Token-Frame oder ein Claim-Frame unbeschädigt angekommen. Um dies festzustellen, überprüft die Access Control Machine als erstes die Variable **frame_control**. Enthält sie den Wert 0x40, ist ein Daten-Frame eingetroffen, bei 0x08 ein Token-Frame und bei 0x00 ein Claim-Frame. Hier zunächst die Struktur des Programm-Fragments (vergleiche auch mit Bild 61):
Ist ein Claim-Frame angekommen, verzweigt die Funktion acm_exe() nach **default** und verlässt dort den case ok-Block. Dies bedeutet, die ACM bleibt im Zustand Idle. Dies ist einsichtig, denn offensichtlich ist gerade eine andere Station dabei, das Token zu erwerben.

```
case ok:
 switch(frame_control)
 {
  case token:
        •
        •
  break;
  case data:
        •
        •
  break;
  default: break;          /* claim */
 }
break;
```

3.1 token

Nachdem die Access Control Machine die MAC Control-Informationen von der Receive Machine empfangen hat, steht ihr in der Variablen dest_station der Wert des Token-Frame-Feldes ns (next station) zur Verfügung. So kann die ACM die Nummer in der Variablen dest_station mit der eigenen Stationsnummer (this_station) vergleichen, und dadurch feststellen, ob der Token-Frame für ihre Station bestimmt ist oder nicht. Betrachten wir dazu das folgende Bild 65.

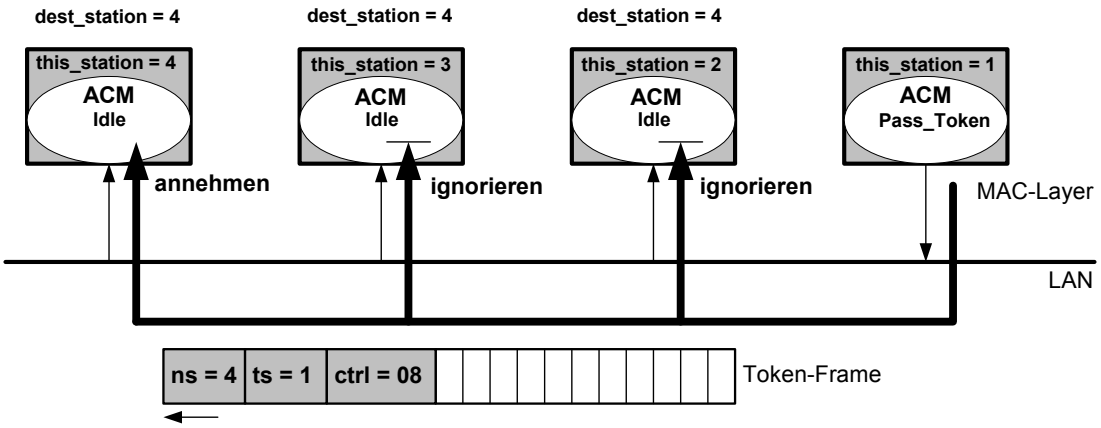
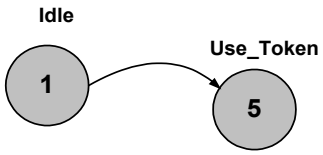


Bild 65: Token-Frame feststellen

Angenommen die Station 1 (this_station=1) ist im Zustand Use_Token. Ihre Sendezeit sei entweder abgelaufen, oder es stehen keine Daten-Frames mehr zur Verfügung. Sie geht deshalb in den Zustand **Pass_Token** und gibt das Token weiter an ihre Nachfolge-Station ns=4. Diese Station (this_station=4) und auch die restlichen Stationen 2 (this_station=2) und 3 (this_station=3) befinden sich im Idle-Zustand. So kommt der Token-Frame bei den Stationen 4, 3 und 2 an, und nur Station 4 erkennt, daß der Token-Frame für sie bestimmt ist. Die Access Control Machine setzt daraufhin mit **set_thtr**(**hi_pri_token_hold_time**) den token hold timer. Sie initialisiert ihn mit der maximalen Sendezeit und veranlasst danach den Wechsel in den Zustand Use_Token.

Anweisungen	Transition

```
case token:
      if(dest_station==this_station)
        {
         set_thtr(hi_pri_token_hold_time);
         acm_status = use_token;
        }
break;
```



3.2 data

Genauso wie beim Empfang des Token-Frames, vergleicht auch hier die Funktion acm_exe() die Variablen **dest_station** mit **this_station**, und stellt fest, ob der empfangene Daten-Frame für ihre Station bestimmt ist.
Wenn **nein**, veranlaßt acm_exe() keine weiteren Aktionen. Die Access Control Machine bleibt im idle-Zustand.
Wenn **ja**, leitet acm_exe() Aktionen ein, die auf den Überlegungen einer gesicherten Datenübertragung basieren.
Hat ein Sender eine Nachricht abgeschickt, hat er keine Gewißheit darüber, ob die Nachricht beim Empfänger tatsächlich ordnungsgemäß angekommen ist.
Diese Gewißheit bekommt er nur, wenn der Empfänger eine Quittung oder eine Antwortnachricht an ihn zurückschickt. Ist sie angekommen, ist der Nachrichten-Zyklus abgeschlossen, und der Sender kann die nächste Übertragung starten.
So muß der Sender solange warten, bis ihn die Quittung erreicht hat. Nun lautet das Token-Prinzip: **„Nur wer das Token hat, darf senden“**.
Da im Augenblick das Token im Besitz des Senders ist, kann der Empfänger keine Qittung schicken. Er muß warten, bis ihn das Token erreicht hat. Dies kann „sehr lange“ dauern, und ist abhängig von der Anzahl der Stationen im LAN. Soll z.B. der Empfänger auf eine Nachricht des Senders innerhalb von 20 ms antworten, kann er die geforderte **Antwortzeit** nicht einhalten.
Betrachten wir dazu folgendes Beispiel. Siehe Bild 66:
Der Rechner 3 (host_id = 3) ist im Besitz des Token. Er arbeitet somit als Sender und schickt dem Empfänger 0 (host_id=0) einen Daten-Frame.

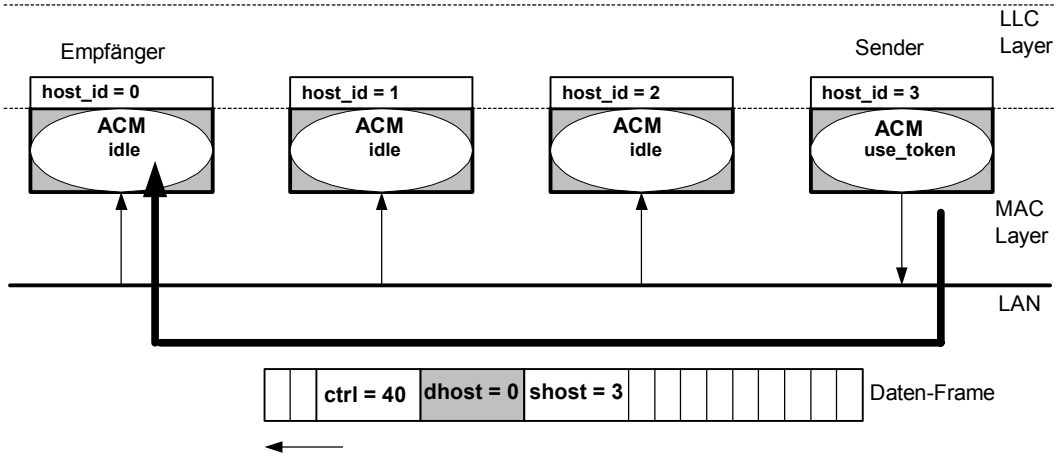


Bild 66: Rechner 3 schickt Rechner 0 einen Daten-Frame

Rechner 0 möchte den empfangenen Daten-Frame quittieren. Nach dem Token-Prinzip muß er auf die Sendeberechtigung (das Token) warten, und kann erst dann die Quittung senden. So wandert das Token von Rechner 3 zum Rechner 2, von dort zum Rechner 1 und trifft schließlich beim Rechner 0 ein. Siehe Bild 67:

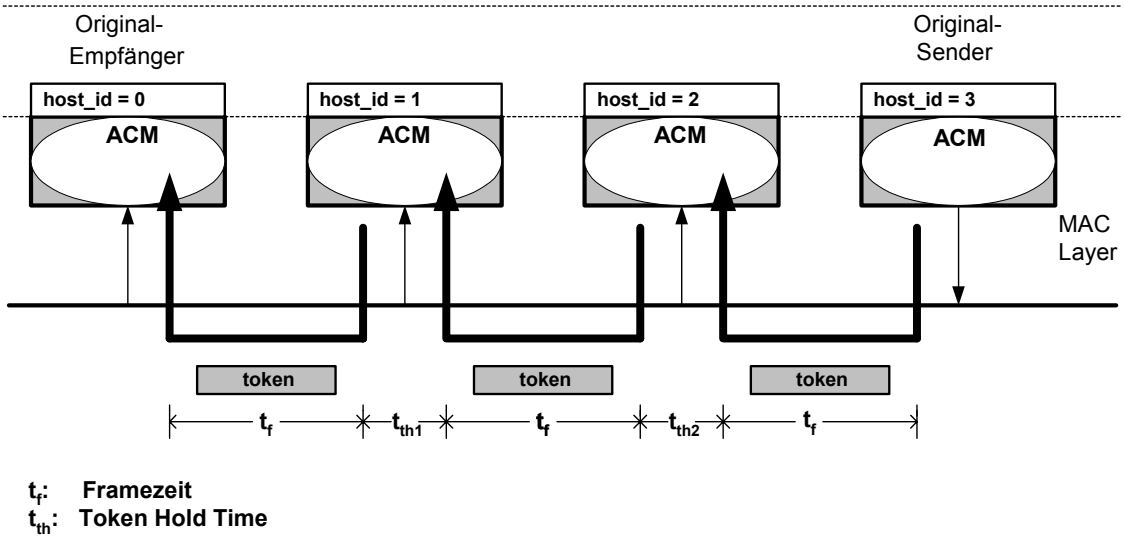


Bild 67: Das Token „wandert“ von Rechner 3 zum Rechner 0

Auf dem Weg vom Rechner 3 zum Rechner 2 braucht das Token die Framezeit t_f. Das gleiche gilt für den Weg vom Rechner 2 zum Rechner 1 und vom Rechner 1 zum Rechner 0. Rechner 2 und Rechner 1 halten das Token maximal die Token Hold Time t_{th} und senden Daten-Frames, wenn welche vorhanden sind. So kann es vorkommen, daß die Token Hold Time nicht vollständig genutzt wird. Ist das Token schließlich nach Ablauf der Zeit

$$t_a = t_f + t_{th2} + t_f + t_{th1} + t_f + \ldots\ldots\ldots$$

beim originalen Empfänger angekommen, wird dieser zum Sender und kann endlich eine Quittung, bzw. einen sogenannten Response-Frame an den originalen Sender zurückschicken. Siehe Bild 68: Es ist einsichtig, daß sich die Token-Ankunftzeit t_a erhöht, je mehr Stationen sich zwischen dem originalen Sender und dem originalen Empfänger befinden.

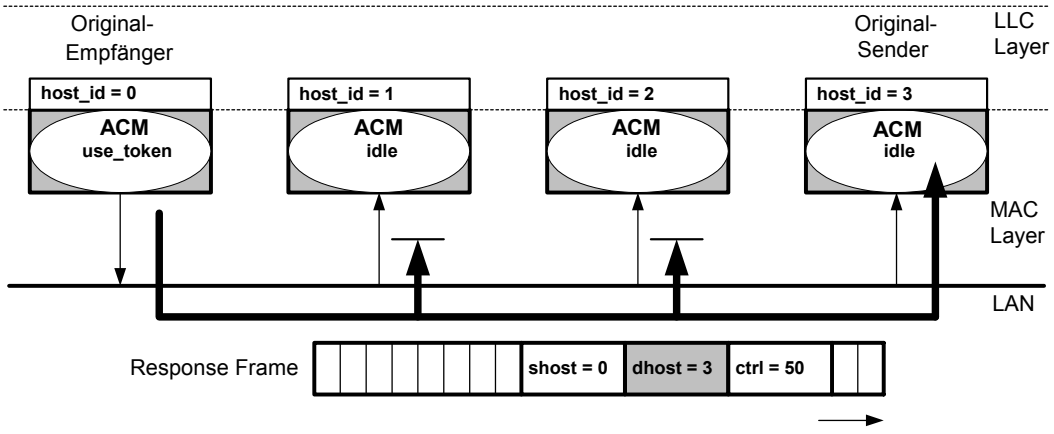


Bild 68: Rechner 0 schickt einen Response-Frame an Rechner 3

Wie lässt sich für den beschriebenen Fall der Nachrichten-Zyklus verkürzen und damit das Realzeitverhalten verbessern? Die einzige Möglichkeit ist, die Token-Ankunftzeit t_a zu verkürzen. Und dies ist nur möglich, wenn das Token-Prinzip außer Kraft gesetzt wird. Schickt ein Sender einem Empfänger einen Daten-Frame, darf der Empfänger, ohne daß er das Token besitzt, unmittelbar einen Response-Frame an den Sender zurückschicken. Auf diese Weise sind die Token Hold Times der dazwischenliegenden Stationen „ausgeschaltet". Wie ein Nachrichten-Zyklus mit einem sogenannten „Immediate Response" aussieht, demonstrieren die folgenden Bilder 69a, 69b:

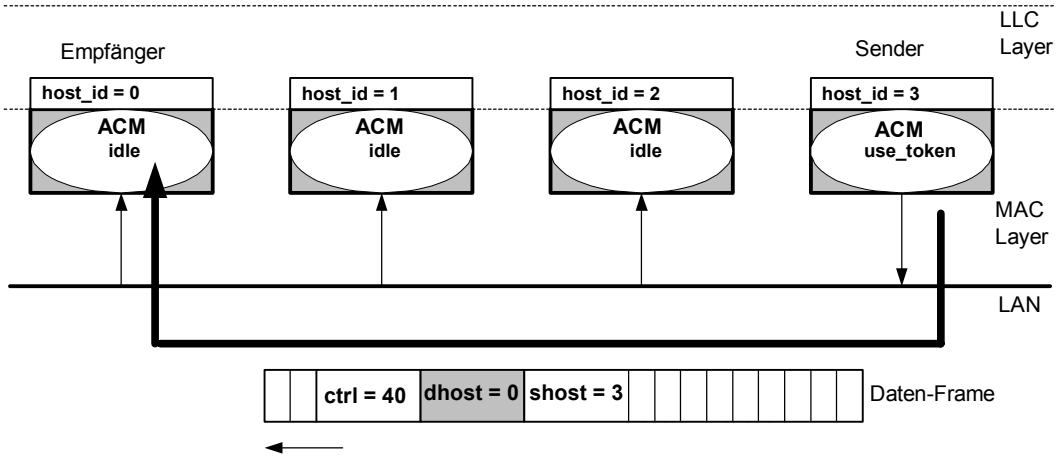


Bild 69a: Rechner 3 schickt Rechner 0 einen Daten-Frame

Wir erkennen, daß sich der Sender im Zustand use_token und der Empfänger im Zustand idle befindet.

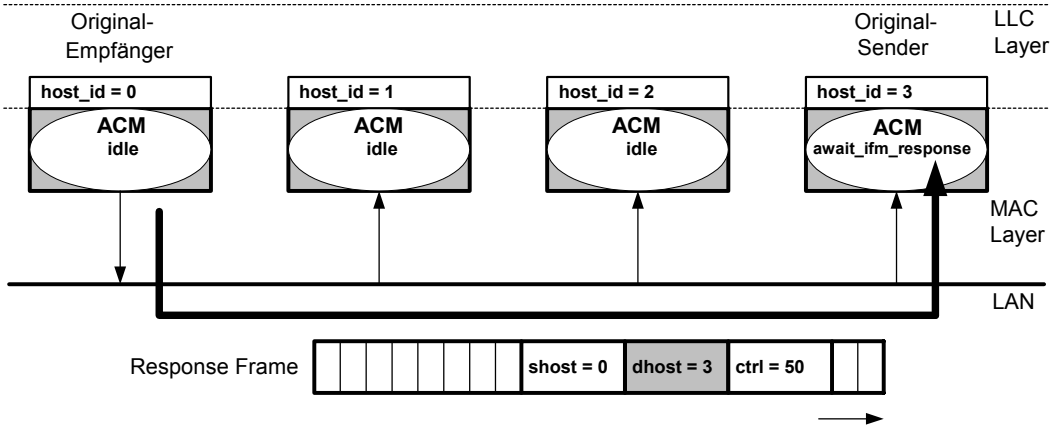


Bild 69b: Immediate Response

Nachdem der Sender den Daten-Frame abgeschickt hat, verläßt er den Zustand use_token und geht über in den Zustand **await_ifm_response** (await **i**nter**f**ace **m**achine response). In diesem Zustand wartet er auf eine Antwort bzw. einen Response-Frame vom Empfänger. Bemerkenswert ist, daß der Empfänger den Response-Frame im Zustand idle an den Sender zurückschickt. **Hinweis**: Wie sich die Access Control Machine im Zustand wait_ifm_response genau verhält, sehen wir etwas später.

Nachdem der Empfänger festgestellt hat, daß der Daten-Frame für ihn bestimmt ist, führt die Funktion acm_exe() weitere Anweisungen aus, die im folgenden Programm-Fragment zusammengefaß sind. Sie veranlassen die Übertragung eines potentiellen Response-Frame zurück an den Sender.

```
case data:
        if(dest_station == this_station)
           {
            ifm_response(source_host,&llc_pdu_pending,mac_pdu);
            if(llc_pdu_pending)
              {
                mac_pdu[ns]   = source_station;
                mac_pdu[ts]   = this_station;
                mac_pdu[ctrl] = response;
                uart_in_sende_modus();
                txm_exe(mac_pdu);
              }
           }
 break;
```

Bild 70: Übertragung eines Response-Frame veranlassen

Der Empfänger identifiziert den Sender aus der erhaltenen Hostnummer **source_host**. Er muß jetzt feststellen, ob für den source_host ein Response-Frame zur Verfügung steht. Der Ort, wo er das erfahren kann, ist die Interface Machine.
Wir erinnern uns, die Interface Machine ist in zwei logische Funktionseinheiten unterteilt: Eine **Sendeeinheit** und eine **Empfangseinheit**. Wie die Empfangseinheit, so ist auch die Sendeeinheit als ein array von Mailboxen realisiert. Die Idee ist, in jedem Rechner (host_id) für jeden Ziel-Rechner (dhost) eine individuelle Mailbox zur Verfügung zu stellen. In jeder dieser Mailboxen halten sich potentielle LLC PDUs für die Ziel-Rechner auf. D. h. in der Mailbox 0 (dhost=0) LLC PDUs für den Ziel-Rechner 0, in der Mailbox 1 (dhost=1) LLC PDUs für den Ziel-Rechner 1 usw.
Hat ein Rechner einen Daten-Frame empfangen, identifiziert seine Access Control Machine den Quell-Rechner. Zur Erinnerung: Die Receive Machine-Funktion sende_acm_event(..) überträgt an die Access Control Machine neben dem MAC Control Feld aus den bekannten sachdienlichen Gründen auch die Variable shost aus dem MAC Daten-Feld. Diese Variable steht der ACM zur Auswertung in der Variablen **source_host** zur Verfügung.
Steht für diesen source_host eine LLC PDU bereit? Die ACM muß nur in der korrespondierenden Mailbox der Interface Machine-Sendeeinheit „nachsehen“: Ist source_host=0 überprüft sie die Mailbox 0 (dhost=0), ist source_host=1 überprüft sie die Mailbox 1 (dhost=1) usw. Steht eine LLC PDU bereit, kann die Access Control Machine einen Response Frame an den Absender zurückschicken.
Betrachten wir dazu das folgende Bild 71. Es demonstriert ein Beispiel und verdeutlicht die Identifikation des source_host und die daraus resultierenden Aktionen.

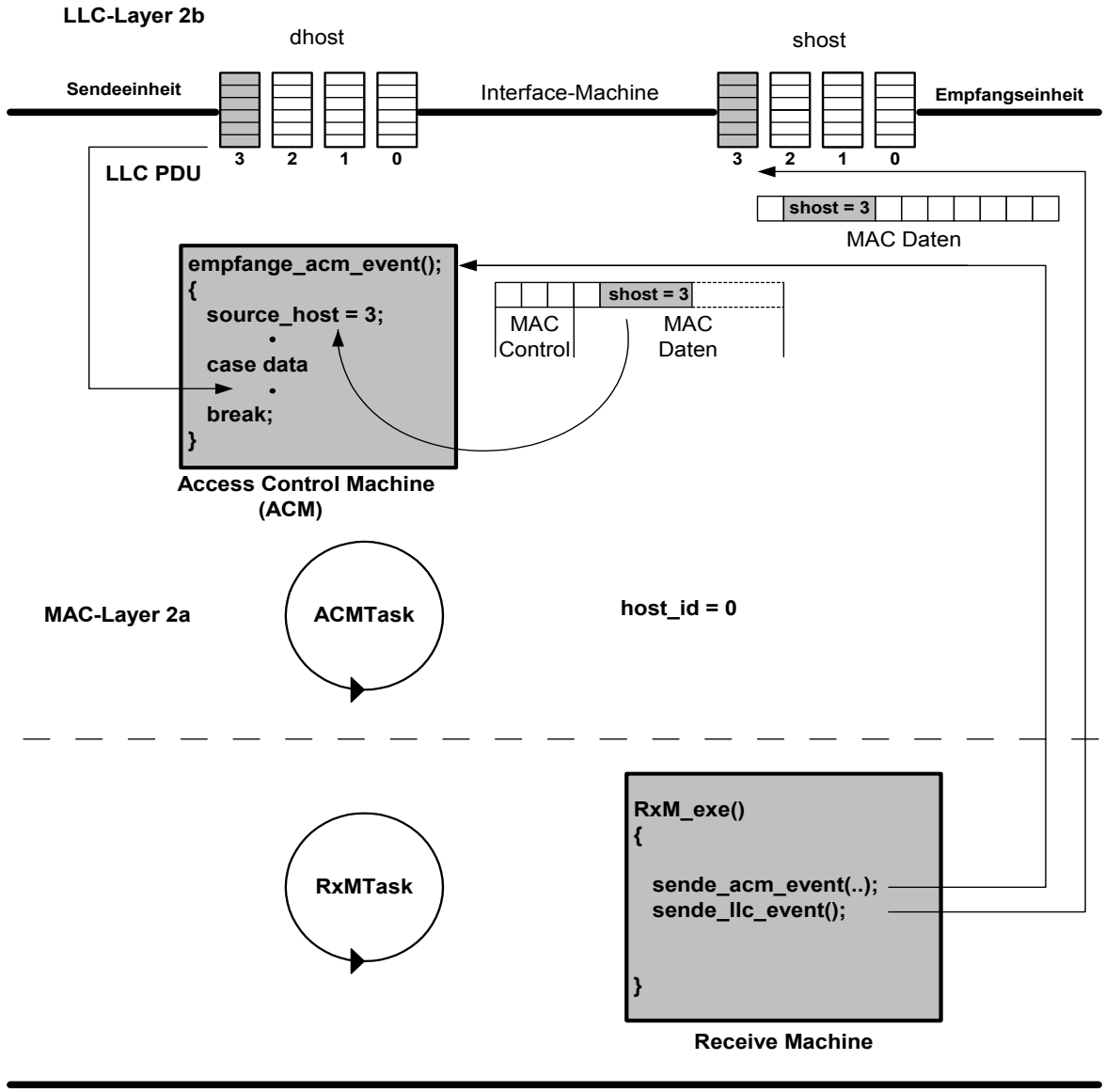


Bild 71: Identifikation des source_host

Der betrachtete Rechner 0 (**host_id=0**) bekommt vom Quell-Rechner 3 (**shost=3**) einen Daten-Frame. Die Receive Machine sendet das MAC Daten-Feld an die Mailbox 3 der Interface Machine-Empfangseinheit und das MAC Control-Feld zusammen mit dem shost-Feld aus den MAC Daten an die Access Control Machine.
Wegen **source_host = shost = 3** prüft die ACM die Mailbox **3** der Interface Machine-**Sendeheit** und stellt dort eine potentielle LLC PDU fest.

1.3.2.1.2 Botschaften von der Interface Machine

Die Überprüfung der betreffenden Mailbox in der Interface Machine-Sendeeinheit veranlaßt die Funktion

- **ifm_response(source_host,&llc_pdu_pending,mac_pdu);**

Vergleiche mit Bild 70. Sie wird mit drei Parametern aufgerufen:

1, **source_host**: Damit identifiziert die Funktion die mit dem Quell-Rechner korrespondierende Mailbox.

2, **&llc_pdu_pending**: Hier trägt die Funktion zur späteren Auswertung den Wert true (1) ein, wenn eine LLC PDU in der Mailbox eingekettet ist oder false (0), wenn die Mailbox leer ist.

3, **mac_pdu**: Dieses array repräsentiert die MAC PDU mit den beiden Feldern MAC Control und MAC Daten. In das MAC Daten-Feld überträgt die Funktion die LLC PDU.

Die Sendeeinheit der Interface Machine wird also benutzt, um eine LLC PDU vom LLC Layer an den MAC Layer weiterzuleiten. Siehe Bild 72:

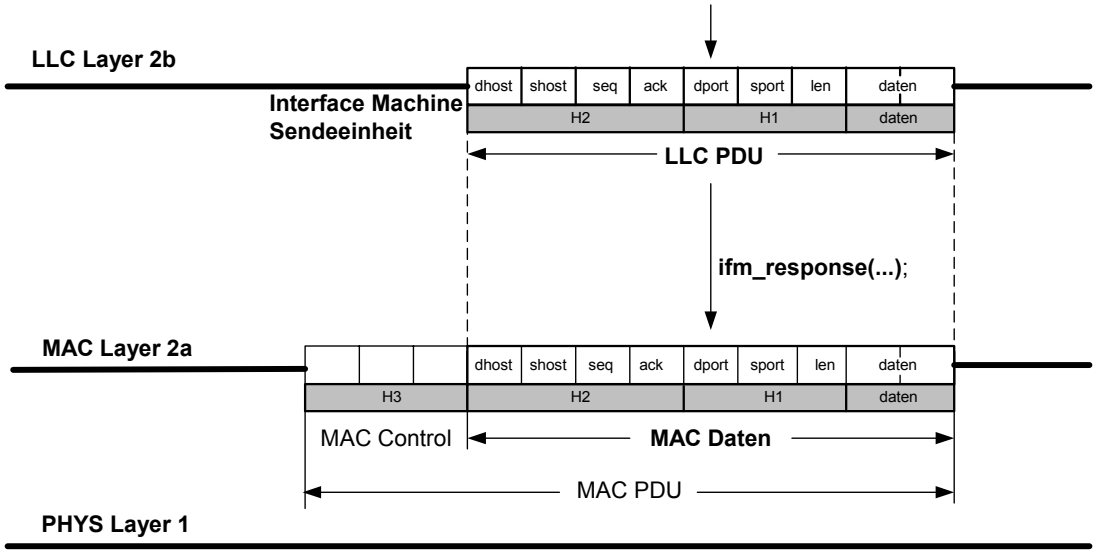


Bild 72: Weiterleiten der LLC PDU vom LLC Layer zum MAC Layer

Wir wollen uns nun ansehen, wie die Funktion ifm_respopnse(...) im Detail arbeitet. Dazu betrachten wir den folgenden Programmcode im Bild 73:

```
void ifm_response(unsigned char source_host,unsigned char *llc_pdu_pending,
                  unsigned char mac_pdu[])
{
 switch(source_host & 0x7f)
 {
  case 0: ifm0_read(llc_pdu_pending,mac_pdu);
  break;
  case 1: ifm1_read(llc_pdu_pending,mac_pdu);
  break;
  case 2: ifm2_read(llc_pdu_pending,mac_pdu);
  break;
  case 3: ifm3_read(llc_pdu_pending,mac_pdu);
  break;
  default: break;
 }
}
```

Bild 73: Betriebssystem-Funktion ifm_response (Datei ifmresp.c)

Zunächst blendet die Funktion in der switch-Anweisung das Control-Bit (Bit 2^7) im Parameter source_host aus. Zur Erinnerung: Ist C=0, sind die **Daten** (2 Bytes) im MAC Daten-Feld gültig. Ist C=1, sind die Daten ungültig, lediglich die seq/ack-Informationen sind von Interesse. Vergleiche auch mit Bild 38.
Der verbleibende Wert in source_host identifiziert den Quell-Rechner, und ifm_response() aktiviert eine der Betriebssystem-Funktionen ifm0_read()....ifm3_read(). Diese Funktionen lesen potentielle LLC PDUs von den mit den Quell-Rechnern korrespondierenden Mailboxen der Interface Machine-Sendeeinheit.
Die Mailboxen sind wie üblich als Queues implementiert; sie bestehen aus einem **ifm_queue_kopf** und einer Anzahl von **llc_pdu**-Elementen. Die Struktur der Queue ist durch die folgenden zwei Datentypen definiert.

```
typedef struct s15
          {
           struct s16 *erster_der_queue;
           struct s16 *letzter_der_queue;
           short  int  anzahl_der_llc_pdus;
          } ifm_queue_kopf_struct;

typedef struct s16
          {
            struct s16 *naechster_in_der_queue;
            unsigned char dhost;
            unsigned char shost;
            unsigned char seq;
            unsigned char ack;
            unsigned char dport;
            unsigned char sport;
            unsigned char len;
            unsigned char daten[2];
           } llc_pdu_struct;
```

Bild 74: Definition der Datentypen ifm_queue_kopf_struct und llc_pdu_struct

Die Datentypen heißen **s15** und **s16**. Beide sind umbenannt, und zwar s15 in ifm_queue_kopf_struct und s16 in llc_pdu_struct.

- Eine Variable des Typs ifm_queue_kopf_struct repräsentiert den **dhost = 0..3** durch ifm_queue_kopf[0]..ifm_queue_kopf[3] und
- eine Variable des Typs llc_pdu_struct repräsentiert eine **LLC PDU** durch unsigned char dhost..unsigned char daten[2].

Die beiden Zeiger erster_der_queue und letzter_der_queue im ifm_queue_kopf zeigen auf das erste und letzte llc_pdu-Element der Queue. Die short int-Variable anzahl_der_llc_pdus zählt die Anzahl der eingeketteten llc_pdu-Elemente, und der Zeiger naechster_in_der_queue in einem llc_pdu-Element zeigt auf das nachfolgende Element.
Damit der LLC Layer eine LLC PDU in eine der Queues einketten kann, muß das Betriebssystem die erforderliche Anzahl von Queues zur Verfügung stellen. So installiert unmittelbar nach dem Systemstart die Betriebssytem-Funktion **init** für jeden dhost einen **ifm_queue_kopf** und für jede Queue ein **llc_pdu**-Element. Siehe Bild 75.
Hinweis: Abhängig von der Art des Sicherungsprotokolls im LLC Layer kann die Anzahl der Elemente angepaßt werden.

```
#define null (void*)              0
#define ifm_queue_max             4
                                  •
                                  •
#define llc_pdu_max               1

                                  •
                                  •
#include<struct7.h>

                                  •
                                  •
ifm_queue_kopf_struct     ifm_queue_kopf[ifm_queue_max];

llc_pdu_struct            llc0_pdu[llc_pdu_max];

llc_pdu_struct            llc1_pdu[llc_pdu_max];
llc_pdu_struct            llc2_pdu[llc_pdu_max];
llc_pdu_struct            llc3_pdu[llc_pdu_max];
                                  •
                                  •
void init(void)
{
          /*  4 ifm_queue-Koepfe installieren */

 for(i=0;i<ifm_queue_max;i++)
   {
    ifm_queue_kopf[i].erster_der_queue        = null;
    ifm_queue_kopf[i].letzter_der_queue       = null;
    ifm_queue_kopf[i].anzahl_der_llc_pdus     = 0;
   }
```

```
for(i=0;i<llc_pdu_max;i++)            /* 1 LLC PDU fuer IFM 0 */
   {
    llc0_pdu[i].naechster_in_der_queue      = null;
    llc0_pdu[i].dhost                       = 0;
    llc0_pdu[i].shost                       = 0;
    llc0_pdu[i].seq                         = 0;
    llc0_pdu[i].ack                         = 0;
    llc0_pdu[i].dport                       = 0;
    llc0_pdu[i].sport                       = 0;
    llc0_pdu[i].len                         = 0;
    llc0_pdu[i].daten[0]                    = 0;
    llc0_pdu[i].daten[1]                    = 0;
   }
 for(i=0;i<llc_pdu_max;i++)            /* 1 LLC PDU fuer IFM 1 */
   {
    llc1_pdu[i].naechster_in_der_queue      = null;
    llc1_pdu[i].dhost                       = 0;
    llc1_pdu[i].shost                       = 0;
    llc1_pdu[i].seq                         = 0;
    llc1_pdu[i].ack                         = 0;
    llc1_pdu[i].dport                       = 0;
    llc1_pdu[i].sport                       = 0;
    llc1_pdu[i].len                         = 0;
    llc1_pdu[i].daten[0]                    = 0;
    llc1_pdu[i].daten[1]                    = 0;
   }
 for(i=0;i<llc_pdu_max;i++)            /* 1 LLC PDU fuer IFM 2 */
   {
    llc2_pdu[i].naechster_in_der_queue      = null;
    llc2_pdu[i].dhost                       = 0;
    llc2_pdu[i].shost                       = 0;
    llc2_pdu[i].seq                         = 0;
    llc2_pdu[i].ack                         = 0;
    llc2_pdu[i].dport                       = 0;
    llc2_pdu[i].sport                       = 0;
    llc2_pdu[i].len                         = 0;
    llc2_pdu[i].daten[0]                    = 0;
    llc2_pdu[i].daten[1]                    = 0;
   }
 for(i=0;i<llc_pdu_max;i++)            /* 1 LLC PDU fuer IFM 3 */
   {
    llc3_pdu[i].naechster_in_der_queue      = null;
    llc3_pdu[i].dhost                       = 0;
    llc3_pdu[i].shost                       = 0;
    llc3_pdu[i].seq                         = 0;
    llc3_pdu[i].ack                         = 0;
    llc3_pdu[i].dport                       = 0;
    llc3_pdu[i].sport                       = 0;
    llc3_pdu[i].len                         = 0;
    llc3_pdu[i].daten[0]                    = 0;
    llc3_pdu[i].daten[1]                    = 0;
   }
}
```

Bild 75: Fragment der Betriebssystem-Funktion init (Datei init.c)

Die Sendeeinheit der Interface-Machine gestaltet sich danach wie folgt. Siehe Bild 76.

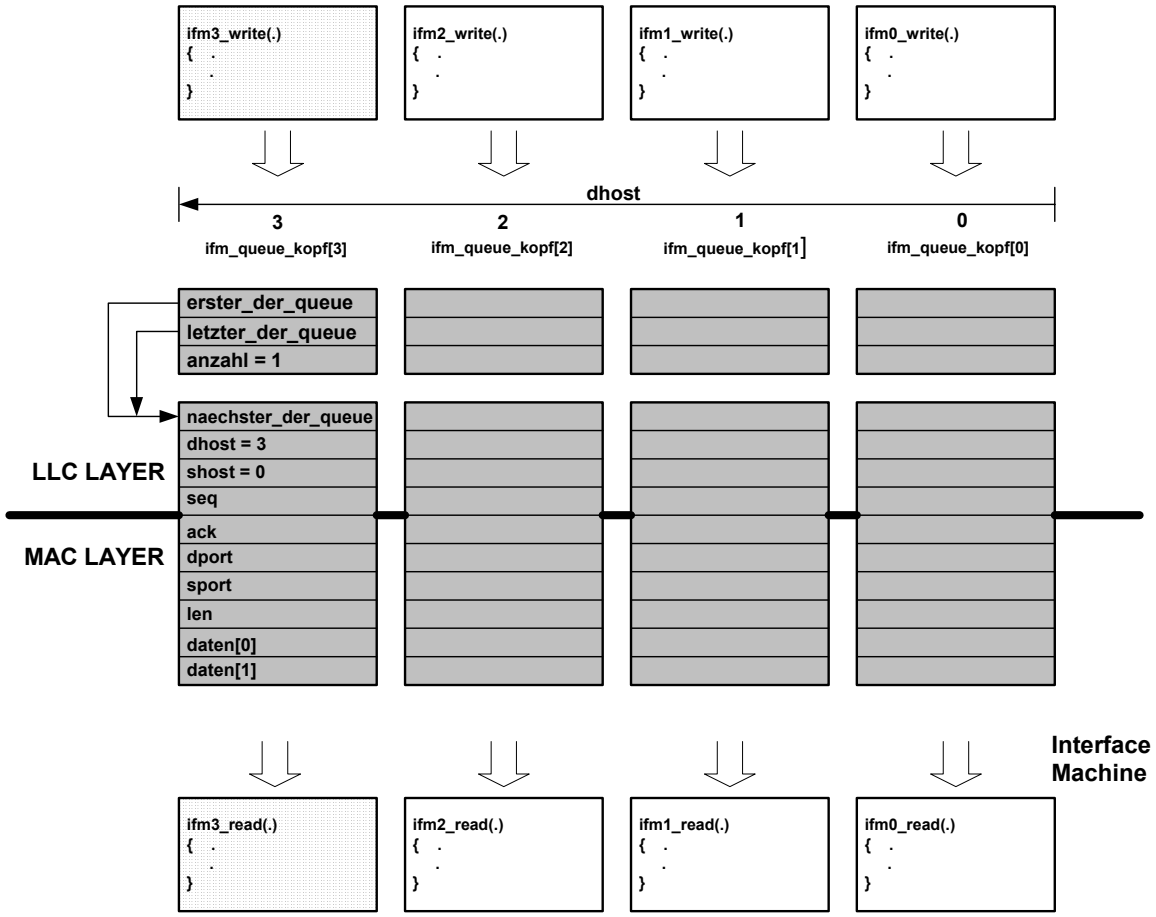


Bild 76: Sendeeinheit der Interface-Machine

Der LLC Layer kettet mittels der Funktionen

- **ifm0_write(.);**
- **ifm1_write(.);**
- **ifm2_write(.);**
- **ifm3_write(.);**

potentielle LLC PDUs als llc_pdu-Elemente in eine der Queues ein. Ist die LLC PDU für den Ziel-Rechner 0 (dhost=0) bestimmt, dann kettet die Funktion ifm**0**_write(.) das llc**0**_pdu-Element in den ifm_queue_kopf[**0**] ein. Ist die LLC PDU für den Ziel-Rechner 1 (dhost=1) bestimmt, dann kettet die Funktion ifm**1**_write(.) das llc**1**_pdu-Element in den ifm_queue_kopf[**1**] ein usw.

Das Bild 76 demonstriert zusätzlich folgenden Fall: Der betrachtete Host hat vom Quell-Rechner 3 einen Daten-Frame empfangen. Dies ist daran zu erkennen, daß die Funktion **ifm3_read(.)** aktiv ist. Sie überprüft den ifm_queue_kopf[3] und stellt dort fest, daß ein llc3_pdu-Element eingekettet ist. Aus den Eintragungen in den Feldern dhost und shost ist weiterhin zu erkennen, daß das Element für den ursprünglichen Quell-Rechner 3 (dhost=3) bestimmt ist, und daß es sich bei dem betrachteten Rechner um den Rechner 0 (shost=0) handelt. So kann die Funktion ifm3_read(.) das eingekettete llc3_pdu-Element in das MAC Daten-Feld übertragen, und auf diese Weise einen Teil der endgültigen MAC PDU bzw. des Response-Frame zu Verfügung stellen (das MAC Control-Feld fehlt noch).
Wie die Funktionen ifm0_read(.)...ifm3_read(.) im Detail arbeiten, verdeutlichen die folgenden Bilder 77, 78, 79 und 80.

```
#include<struct7.h>

#define true                    1
#define false                   0
#define ifm_queue_max     4

extern ifm_queue_kopf_struct     ifm_queue_kopf[ifm_queue_max];
extern unsigned char                 llc_pdu_pending;

void ifm0_read(unsigned char *llc_pdu_pending,unsigned char mac_pdu[])
{
 ifm_queue_kopf_struct   *ifm_queue_kopf_adr;
 llc_pdu_struct              *llc_pdu_adr;

 if(ifm_queue_kopf[0].anzahl_der_llc_pdus == 0)
   *llc_pdu_pending = false;
 else
  {
   *llc_pdu_pending = true;

   ifm_queue_kopf_adr = &ifm_queue_kopf[0];
   llc_pdu_adr             = ifm_queue_kopf_adr -> erster_der_queue;

        /*  llc_pdu empfangen */

   mac_pdu[3]  = llc_pdu_adr->dhost;
   mac_pdu[4]  = llc_pdu_adr->shost;
   mac_pdu[5]  = llc_pdu_adr->seq;
   mac_pdu[6]  = llc_pdu_adr->ack;
   mac_pdu[7]  = llc_pdu_adr->dport;
   mac_pdu[8]  = llc_pdu_adr->sport;
   mac_pdu[9]  = llc_pdu_adr->len;
   mac_pdu[10] = llc_pdu_adr->daten[0];
   mac_pdu[11] = llc_pdu_adr->daten[1];

   ausketten(&ifm_queue_kopf[0]);
  }
}
```

Bild 77: Betriebssystem-Funktion ifm0_read; Datei (ifm0read.c)

```
#include<struct7.h>

#define true                    1
#define false                   0
#define ifm_queue_max      4

extern ifm_queue_kopf_struct    ifm_queue_kopf[ifm_queue_max];
extern unsigned char                llc_pdu_pending;

void ifm1_read(unsigned char *llc_pdu_pending,unsigned char mac_pdu[])
{
 ifm_queue_kopf_struct   *ifm_queue_kopf_adr;
 llc_pdu_struct               *llc_pdu_adr;

 if(ifm_queue_kopf[1].anzahl_der_llc_pdus == 0)
   *llc_pdu_pending = false;
 else
  {
   *llc_pdu_pending = true;

   ifm_queue_kopf_adr = &ifm_queue_kopf[1];
   llc_pdu_adr              = ifm_queue_kopf_adr -> erster_der_queue;

         /*  llc_pdu empfangen */

   mac_pdu[3]  = llc_pdu_adr->dhost;
   mac_pdu[4]  = llc_pdu_adr->shost;
   mac_pdu[5]  = llc_pdu_adr->seq;
   mac_pdu[6]  = llc_pdu_adr->ack;
   mac_pdu[7]  = llc_pdu_adr->dport;
   mac_pdu[8]  = llc_pdu_adr->sport;
   mac_pdu[9]  = llc_pdu_adr->len;
   mac_pdu[10] = llc_pdu_adr->daten[0];
   mac_pdu[11] = llc_pdu_adr->daten[1];

   ausketten(&ifm_queue_kopf[1]);
  }
}
```

Bild 78: Betriebssystem-Funktion ifm1_read; Datei (ifm1read.c)

```
#include<struct7.h>

#define true                    1
#define false                   0
#define ifm_queue_max      4

extern ifm_queue_kopf_struct    ifm_queue_kopf[ifm_queue_max];
extern unsigned char                llc_pdu_pending;

void ifm2_read(unsigned char *llc_pdu_pending,unsigned char mac_pdu[])
{
 ifm_queue_kopf_struct    *ifm_queue_kopf_adr;
 llc_pdu_struct               *llc_pdu_adr;

 if(ifm_queue_kopf[2].anzahl_der_llc_pdus == 0)
   *llc_pdu_pending = false;
 else
  {
   *llc_pdu_pending = true;

   ifm_queue_kopf_adr = &ifm_queue_kopf[2];
   llc_pdu_adr              = ifm_queue_kopf_adr -> erster_der_queue;

         /*  llc_pdu empfangen */

   mac_pdu[3]  = llc_pdu_adr->dhost;
   mac_pdu[4]  = llc_pdu_adr->shost;
   mac_pdu[5]  = llc_pdu_adr->seq;
   mac_pdu[6]  = llc_pdu_adr->ack;
   mac_pdu[7]  = llc_pdu_adr->dport;
   mac_pdu[8]  = llc_pdu_adr->sport;
   mac_pdu[9]  = llc_pdu_adr->len;
   mac_pdu[10] = llc_pdu_adr->daten[0];
   mac_pdu[11] = llc_pdu_adr->daten[1];

   ausketten(&ifm_queue_kopf[2]);
  }
}
```

Bild 79: Betriebssystem-Funktion ifm2_read; Datei (ifm2read.c)

```
#include<struct7.h>

#define true                    1
#define false                   0
#define ifm_queue_max      4

extern ifm_queue_kopf_struct    ifm_queue_kopf[ifm_queue_max];
extern unsigned char                llc_pdu_pending;

void ifm3_read(unsigned char *llc_pdu_pending,unsigned char mac_pdu[])
{
 ifm_queue_kopf_struct    *ifm_queue_kopf_adr;
 llc_pdu_struct               *llc_pdu_adr;

 if(ifm_queue_kopf[3].anzahl_der_llc_pdus == 0)
   *llc_pdu_pending = false;
 else
  {
   *llc_pdu_pending = true;

   ifm_queue_kopf_adr = &ifm_queue_kopf[3];
   llc_pdu_adr              = ifm_queue_kopf_adr -> erster_der_queue;

         /*  llc_pdu empfangen */

   mac_pdu[3]  = llc_pdu_adr->dhost;
   mac_pdu[4]  = llc_pdu_adr->shost;
   mac_pdu[5]  = llc_pdu_adr->seq;
   mac_pdu[6]  = llc_pdu_adr->ack;
   mac_pdu[7]  = llc_pdu_adr->dport;
   mac_pdu[8]  = llc_pdu_adr->sport;
   mac_pdu[9]  = llc_pdu_adr->len;
   mac_pdu[10] = llc_pdu_adr->daten[0];
   mac_pdu[11] = llc_pdu_adr->daten[1];

   ausketten(&ifm_queue_kopf[3]);
  }
}
```

Bild 80: Betriebssystem-Funktion ifm3_read; Datei (ifm3read.c)

Alle Funktionen werden mit der Adresse der unsigned char-Zeigervariablen **llc_pdu_pending** und der Adresse des arrays **mac_pdu[]** aufgerufen. Sie überprüfen als erstes die **anzahl_der_llc_pdus** in den IFM-Queue-Köpfen ifm_queue_kopf[0] oder ifm_queue_kopf[1]...ifm_queue_kopf[3].
Ist die Anzahl 0, speichern die Funktionen in *llc_pdu_pending den Wert **false=0**; im anderen Fall den Wert **true=1**. So meldet diese Variable, ob ein llc_pdu-Element in der betreffenden Queue eingekettet ist oder nicht.
Ist ein Element eingekettet, überträgt die aktive Funktion die Felder des llc_pdu-Elements dhost.....daten[1] in das MAC Daten-Feld der mac_pdu[3].....mac_pdu[11] und kettet anschließend das llc_pdu-Element aus der Queue aus. So ist die Queue wieder leer und das Sicherungsprotokoll des LLC Layers kann ein neues llc_pdu-Element in die Queue einketten.
Die Funktion ifm_response(...) mit seinen Unterfunktionen ifm0_read(..)..ifm3_read(..) ist nun erklärt. Was geschieht als nächstes? Betrachten wir dazu Bild 70. Es ist hier nochmals dargestellt.

```
case data:
        if(dest_station == this_station)
           {
            ifm_response(source_host,&llc_pdu_pending,mac_pdu);
            if(llc_pdu_pending)
              {
                mac_pdu[ns]   = source_station;
                mac_pdu[ts]   = this_station;
                mac_pdu[ctrl] = response;
                uart_in_sende_modus();
                txm_exe(mac_pdu);
              }
           }
 break;
```

Die Access Control Machine wertet die Variable llc_pdu_pending aus. Enthält sie den Wert false, stehen keine gültigen Daten im Datenfeld (MAC Daten) der mac_pdu[] zur Verfügung. So verläßt die Funktion acm_exe() den case data-Block und beläßt die Access Control Machine im idle-Zustand. Dort wartet sie auf neue Ereignisse.
Enthält sie den Wert true, stehen gültige Daten im Datenfeld der mac_pdu[] bereit. Um den Response-Frame zu vervollständigen, muß lediglich das Controlfeld (MAC Control) der mac_pdu[] noch „aufgefüllt“ werden. So bekommt mac_pdu[ns] = mac_pdu[0] die Nummer der source_station. Damit ist die ursprüngliche Sendestation identifiziert. Das Element mac_pdu[ts] = mac_pdu[1] bekommt die Nummer der antwortenden Station this_station und weil es sich um einen Response-Frame handelt, bekommt mac_pdu[ctrl] = mac_pdu[2] den Wert 0x50 (response). Der Response-Frame ist jetzt komplett, und die Access Control Machine kann ihn zur Übertragung an die ursprüngliche Sendestation an die **Transmit Machine (TxM)** weiterleiten.
Zu diesem Zweck schaltet die ACM den UART zunächst in den Sendemodus, und benutzt dafür die Funktion **uart_in_sende_modus();**

Den Programmcode dieser Funktion zeigt Bild 81.

```
#include<i86.h>

#define ir9_disable      0x02
#define ier1             0x381
#define mcr1             0x384

extern unsigned short int    ocw1_reg_slave;
extern unsigned char         ocw1_slave;

uart_in_sende_modus()
{
 ocw1_slave |= ir9_disable;
 outbyte(ocw1_reg_slave,ocw1_slave);     /*          IR9 sperren            */
 outbyte(ier1,0x00);                     /*  FIFO-Interrupt sperren         */
 outbyte(mcr1,9);                        /*    UART in Sende-Modus          */
}
```

Bild 81: Betriebssystem-Funktion uart_in_sende_modus (Datei sendmod.c)

Zunächst wird eine potentielle Interrupt-Anforderung am **IR9**-Eingang der Interrupt-Kaskade gesperrt (siehe Bild 13). Dies geschieht dadurch, daß das entsprechende Masken-Bit im OCW1-Register (**O**peration **C**ode **W**ord) des Interrupt-Controllers 1 (Slave) mit **ocw1_slave |= ir9_disable**; und **outbyte(ocw1_reg_slave,ocw1_slave)**; gesetzt wird. Dabei ist ocw1_reg_slave die Adresse des OCW1-Registers , namlich **0xa1**und ocw1_slave ist der Initialisierungswert (beides ist extern in der Betriebssystem-Funktion ioinit() definiert). Anschließend wird mit **outbyte(ier1,0x00)**; das **I**nterrupt **E**nable **R**egister des UART (Adresse ist 0x381) mit dem Wert 0 geladen. Damit kann sein RxRDY-Ausgang (Receiver Ready) keine Interrupt-Anforderung (Low-Pegel) generieren. Zum Schluß wird mit outbyte(mcr1,9); das Modem Control Register des UART (Adresse ist 0x384) mit dem Wert 9 geladen. Damit wird sein RxD-Eingang (Receive Data) gesperrt und sein TxD-Ausgang (Transmit Data) freigegeben. Der UART ist im **Sende**-Modus. Vergleiche mit Bild 5: Halbduplex-Kommunikation mit RS485.
Für die Übertragung des Response-Frame benutzt die Access Control Machine die Funktion **txm_exe(mac_pdu);** die mit der Adresse des arrays mac_pdu[] aufgerufen wird. Wie die Funktion arbeitet, wird im Kapitel „Die Transmit Machine“ beschrieben.
Die Funktion acm_exe() verläßt jetzt den case data-Block und beläßt die Access Control Machine im idle-Zustand. In diesem Zustand wartet sie auf neue Ereignisse.

1.3.2.1.3 Der Zustand Use Token

Diesen Zustand erreicht die ACM durch Übergänge aus dem Zuständen **Idle** oder **Await_ifm_response**. Siehe Bild 59: ACM Finite State Machine-Diagramm. In diesem Zustand besitzt die ACM das Token und die Station ist berechtigt Daten-Frames zu senden. Wo findet die ACM die Sende-Daten? Wenn welche vorhanden sind, dann sind sie als LLC PDUs in den ifm_queues der **Interface Machine-Sendeeinheit** eingekettet.
Wir wissen, daß die ifm_queue-Köpfe (ifm_queue_kopf[0]...ifm_queue_kopf[3]) die Ziel-Rechner (dhosts) repräsentieren. In welcher Queue soll die ACM „nachsehen“ , um eingekettete LLC PDUs festzustellen? Die einfache Antwort lautet: In **allen**, denn es gibt keinen sachdienlichen Grund den einen oder anderen Ziel-Rechner zu bevorzugen. Und so suchen wir nach einen Algorithmus, der dieses Vorhaben realisiert.

Als Lösung bietet sich die **round robin**-Methode an. In einer for-Schleife, deren Laufvariable i im Bereich 0<=i<ifm_queue_max liegt, wird reihum in allen ifm_queue-Köpfen der Interface Machine-Sendeeinheit die **anzahl_der_llc_pdus** überprüft.

- Ist die erste Queue gefunden, in die eine LLC PDU eingekettet ist *(anzahl_der_llc_pdus != 0),*
  wird die for-Schleife verlassen und das Element 6 im access class-Vektor sending[max_access_class+1] gleich 1 bzw. true gesetzt: send_pending[6] = true.
  Diese Meldung wird die Access Control Machine später veranlassen, die betreffende LLC PDU in eine MAC PDU zu „packen“ und als Daten-Frame zu übertragen.

- Sind während des Ablaufs der for-Schleife nur **leere** Queues festgestellt worden *(anzahl_der_llc_pdus == 0),*
  wird send_pending[6] gleich 0 bzw. false gesetzt: send_pending[6] = false.
  Diese Meldung wird die Access Control Machine später veranlassen, das Token an ihre Nachfolge-Station weiterzuleiten.

Betrachten wir nun einige Szenarien, die den Algorithmus verdeutlichen. Es liege als erstes folgende Situation vor: Der LLC Layer hat in die ifm_queue-Köpfe 1 und 3 (ifm_queue_kopf[1] und ifm_queue_kopf[3]) je eine LLC PDU eingekettet.

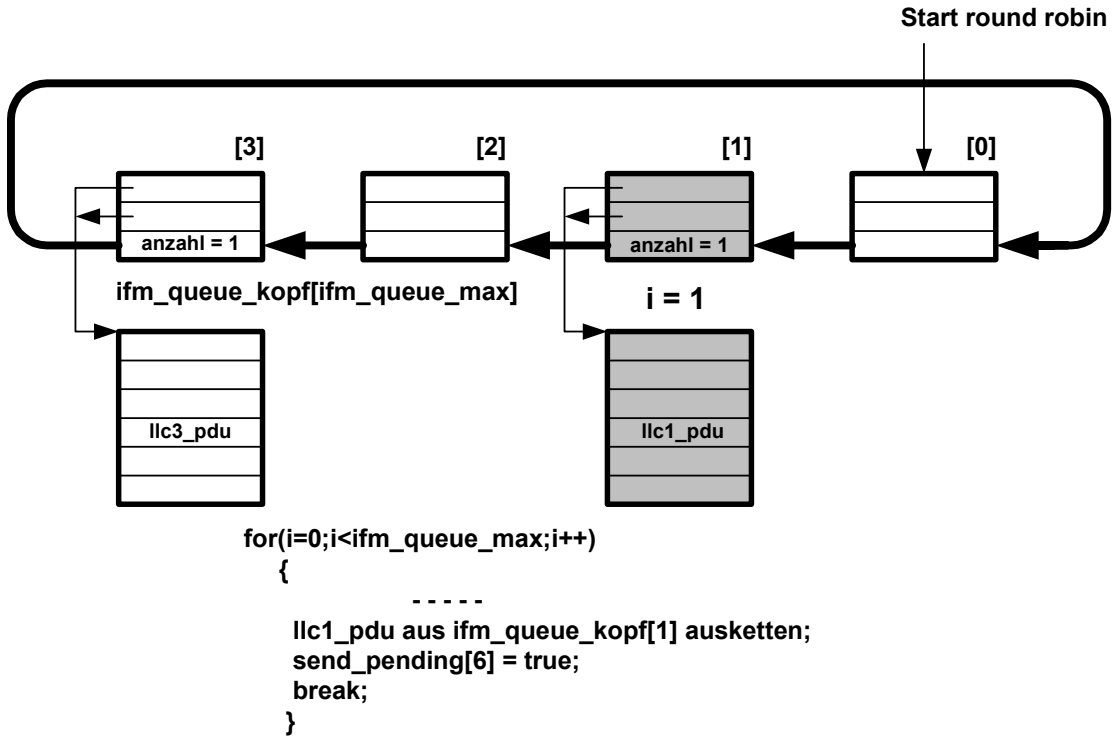


Bild 82: Round Robin: 1ter Durchlauf

Der Algorithmus beginnt mit der Ausführung der for-Schleife, und stellt bei Index i=1 die erste **nicht leere** Queue fest. Er kettet die llc1_pdu aus, verläßt mit **break** die Schleife, und veranlaßt danach die Übertragung der LLC PDU an den Ziel-Rechner 1 (dhost=1). Anschließend verläßt die Acces Control Machine den Zustand Use_Token, geht in den Zustand **Await_ifm_response** und wartet in diesem Zustand auf die Ankunft eines Response-Frame. Vergleiche mit den Bildern 69a und 69b.

Uns ist bewußt, daß in Wahrheit nicht die Access Control Machine wartet, sondern die **ACMTask**. Die ACM wertet lediglich die eingetroffenen Ereignisse aus. Während sich die ACMTask im Wartezustand befindet, nehmen andere Tasks den Prozessor in Beschlag. So kann es vorkommen, daß währenddessen die Tasks des LLC Layers aktiv sind und weitere LLC PDUs in die Interface Machine-Sendeeinheit einketten.
Die ACMTask verläßt nach Ablauf einer begrenzten Wartezeit oder nach Ankunft eines Response-Frame ihren Wartezustand, so daß die Access Control Machine ihre Arbeit wieder fortsetzen kann. Ist der Response-Frame eingetroffen, kehrt sie zurück in den Zustand **Use_Token** und sie beginnt dort erneut mit der Durchsuchung der ifm_queue-Köpfe. Dabei kann es sein, daß sie jetzt zum Beispiel folgende veränderte Situation vorfindet.
Siehe Bild 83.

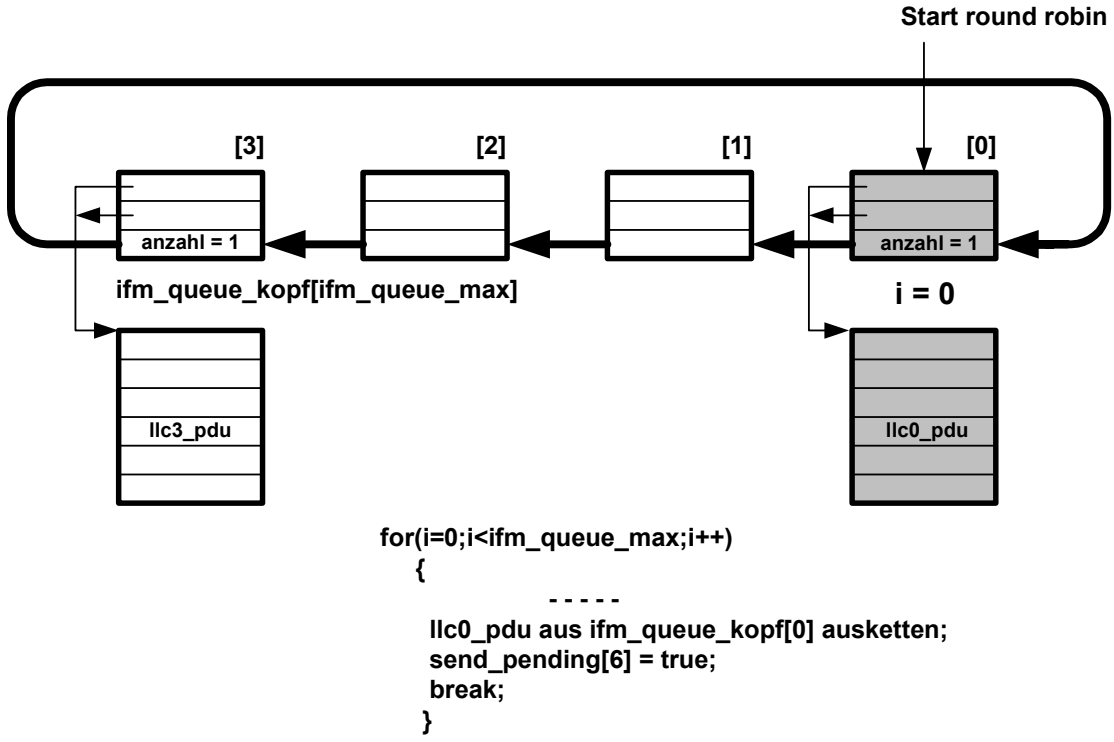


Bild 83: Round Robin; 2ter Durchlauf

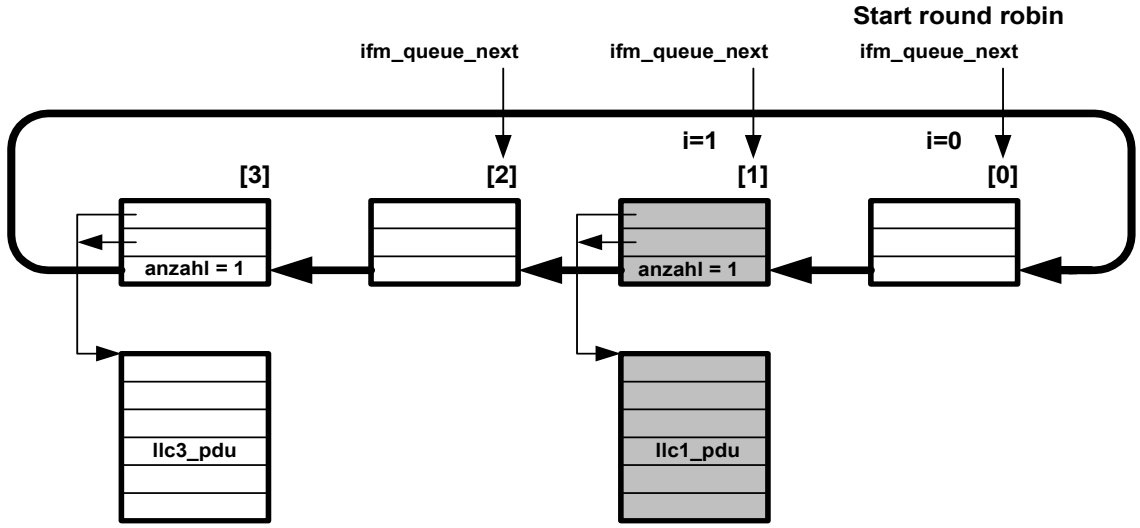
Offensichtlich hat während der ACMTask-Warteperiode der LLC Layer in die Queue bei Index = 0 eine LLC PDU (llc0_pdu) eingekettet. Dies stellt der round robin-Algorithmus bei der Ausführung der for-Schleife fest. Wie üblich kettet er die llc0_pdu aus, verläßt mit **break** die for-Schleife und veranlaßt die Übertragung der LLC PDU an den Ziel-Rechner 0 (dhost=0). Obwohl die llc3_pdu im ifm_queue_kopf[3] schon länger als die llc0_pdu auf die Übertragung wartet, bleibt sie **unbeachtet**. So begünstigt der Algorithmus „Vordrängler“. Daher suchen wir nach einen leistungsfähigeren Algorithmus, der auch das Konzept der Gerechtigkeit beachtet.

Mit Hilfe eines separaten Indizierers, wir nennen ihn ifm_queue_next, ist dieses Ziel erreichbar.
So indiziert jetzt nicht mehr die Laufvariable i die ifm_queue-Köpfe, sondern die Variable ifm_queue_next: **ifm_queue_kopf[ifm_queue_next]**

Die for-Schleife wird jetzt neben anderem als „Antrieb“ für den neuen Indizierer benutzt, und schaltet ihn von Queue zu Queue weiter. Wird während der Ausführung der for-Schleife eine LLC PDU festgestellt,

- wird sie mit ausketten(&ifm_queue_kopf[ifm_queue_next]); aus der Queue entfernt,
- danach wird mit ifm_queue_next++; die nächst folgende Queue indiziert und
- schließlich mit break; die for-Schleife verlassen.

Nimmt später die Access Control Machine ihre round robin-Durchsuchung in der Interface Machine-Sendeeinheit wieder auf, werden jetzt die Queues ab ifm_queue_next = ? indiziert und nicht mehr ab i=0. So sorgt die for-Schleife zum einen dafür, daß im Bedarfsfall alle ifm_queue-Köpfe durchsucht werden (i=0;i<ifm_queue_max;i++) und zum anderen, daß dies mit Hilfe von ifm_queue_next reihum geschieht. Auf diese Weise wird kein „Vordrängler“ bevorzugt. Betrachten wir dazu einige Szenarien. Siehe Bilder 84 und 85.




```
for(i=0;i<ifm_queue_max;i++)
   {
                - - - - -
    ausketten(&ifm_queue_kopf[1]);
    send_pending[6] = true;
    ifm_queue_next++;
    break;
   }
```

Bild 84: Verbessertes round robin; 1te Durchsuchung

In die LLC Queues 1 und 3 seien LLC PDUs eingekettet. Der Indizierer ifm_queue_next habe den anfänglichen Wert 0. Die for-Schleife beginnt ihre Arbeit mit i=0 und **ifm_queue_next=0** und stellt am ifm_queue_kopf[ifm_queue_next] = ifm_queue_kopf[0] eine leere Queue fest. Am Ende des ersten Durchlaufs bekommt der Indizierer wegen ifm_queue_next++; den Wert 1.

Es beginnt ein zweiter Durchlauf mit i=1, und es wird jetzt mit ifm_queue_next=1 am ifm_queue_kopf[1] eine nicht leere Queue festgestellt. Die entsprechende llc1_pdu wird ausgekettet, danach ifm_queue_next inkrementert und schließlich die for-Schleife bei i=1 mit break verlassen. Die ACMTask geht in den Wartezustand, und der Indizierer zeigt wegen ifm_queue_next=2 zum nachfolgenden ifm_queue_kopf[2]. Siehe Bild 84.
Nach Ablauf der Wartezeit im Zustand Await_ifm_response kehrt die Access Control Machine wieder zurück in den Zustand Use_Token und beginnt von neuem mit der Ausführung der for-Schleife. Doch zwischenzeitlich ist der LLC Layer aktiv gewesen und hat in die LLC Queue 0 die llc0_pdu eingekettet. Siehe Bild 85.

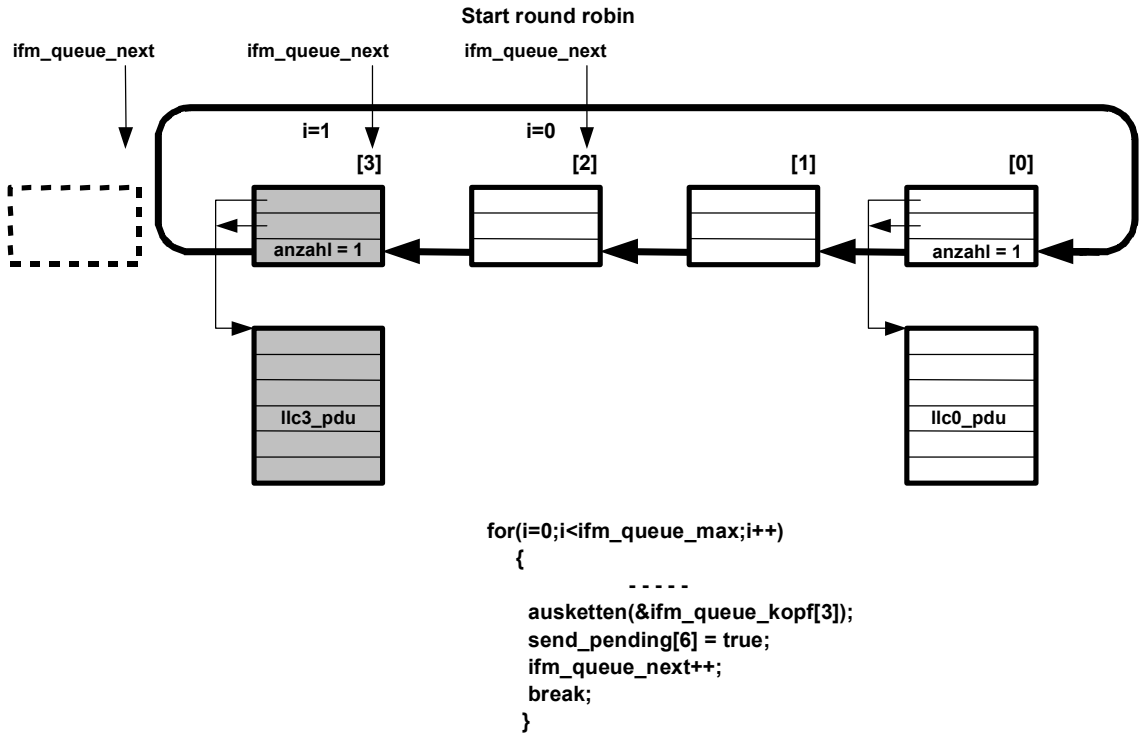


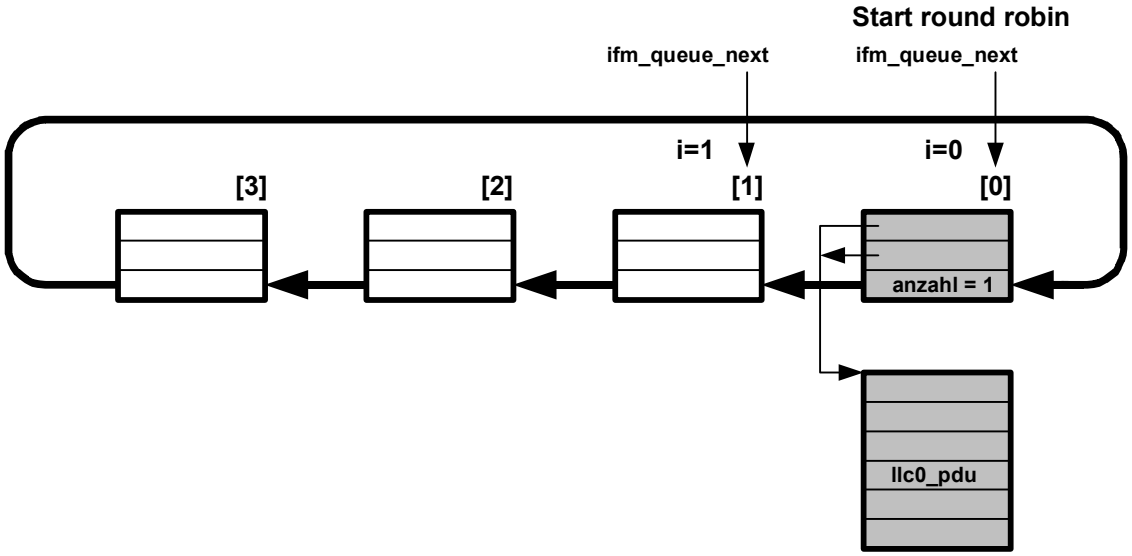
Bild 85: Verbessertes round robin; 2te Durchsuchung

Die for-Schleife beginnt ihre Arbeit wie üblich mit i=0, aber jetzt mit ifm_queue_next=2 und stellt am ifm_queue_kopf[2] eine leere Queue fest. Am Ende des ersten Durchlaufs bekommt der Indizierer wegen ifm_queue_next++; den Wert 3. So beginnt ein nächster Schleifendurchlauf mit i=1 und es wird wegen ifm_queue_next=3 am ifm_queue_kopf[3] eine nicht leere Queue festgestellt. Die entsprechende llc3_pdu wird ausgekettet, danach ifm_queue_next inkrementiert und schließlich die for-Schleife mit break verlassen. Die ACMTask geht wieder in den Wartezustand. Doch welchen Wert hat jetzt der Indizierer? Wegen ifm_queue_next++; ist der Wert 4, und somit indiziert die Variable den ifm_queue_kopf[4], der allerdings nicht initialisiert ist. Was ist zu tun? Ein kleiner Algorithmus muß dafür sorgen, daß das array ifm_queue_kopf[ifm_queue_max] zu einem Ring geschlosen wird. Mit anderen Worten, wenn ifm_queue_next = ifm_queue_max ist, muß ifm_queue_next=0 werden, und damit auf den Beginn des arrays, nämlich auf den ifm_queue_kopf[0] zeigen.

Damit sieht der Algorithmus folgendermaßen aus:

```
if(ifm_queue_next == ifm_queue_max)
   ifm_queue_next   = 0;
```

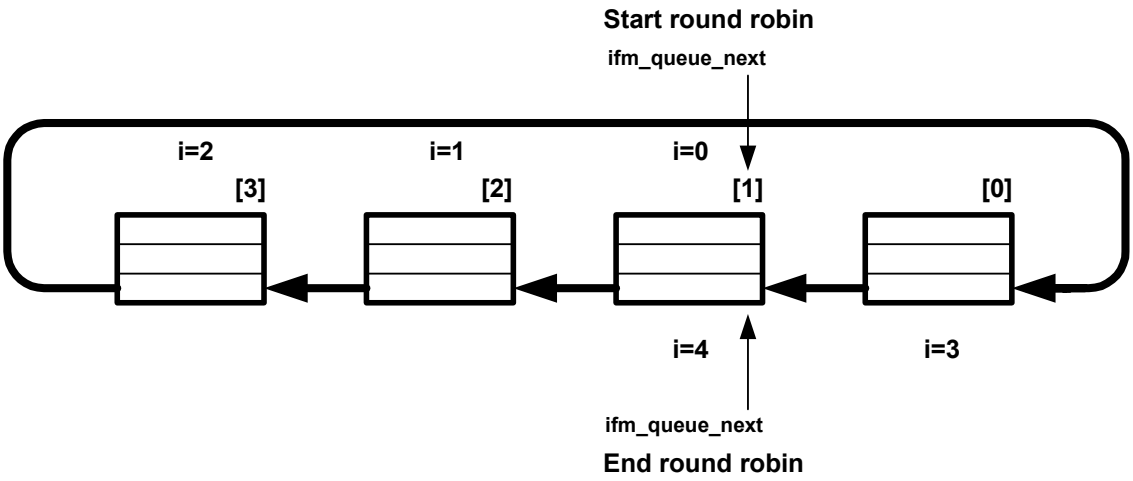
An welche Stelle soll er platziert werden? Es kann nur der Anfang der for-Schleife sein. Wenn
die Access Control Machine die Ausführung der for-Schleife beginnt, schließt, wenn notwendig, der Algorithmus das array ifm_queue_kopf[ifm_queue_max] zu einen Ring. So ist auf diese Weise round robin realisiert. Die Wirkung erkennt man am folgenden Bild 86, wenn zum 3te mal die Interface Machine-Sendeeinheit durchsucht wird.




```
for(i=0;i<ifm_queue_max;i++)
   {
    if(ifm_queue_next == ifm_queue_max)
       ifm_queue_next  =  0;
             - - - - -
    ausketten(&ifm_queue_kopf[0]);
    send_pending[6] = true;
    ifm_queue_next++;
    break;
   }
```

Bild 86: Verbessertes round robin; 3te Durchsuchung

Die Variable ifm_queue_next bekommt als erstes den Wert 0 und indiziert damit den llc_queue_kopf[0]. Dort ist die llc0_pdu eingekettet. Wie üblich wird sie aus der Queue entfernt und die for-Schleife verlassen. Der Indizierer ifm_queue_next hat den Wert 1 bekommen und zeigt auf den ifm_queue_kopf[1]. Was geschieht, wenn der LLC Layer während der Wartezeit der ACMTask keine LLC PDUs in die Interface Machine-Sendeeinheit eingekettet hat? Die Antwort visualisiert das folgende Bild 87.




```
for(i=0;i<ifm_queue_max;i++)
   {
    if(ifm_queue_next == ifm_queue_max)
      ifm_queue_next    = 0;
              - - - - -
    send_pending[6] = false;
    ifm_queue_next++;
   }
```

Bild 87: Verbessertes round robin; 4te Durchsuchung

In der nächsten Durchsuchung der ifm_queue-Köpfe beginnt die for-Schleife bei i=0 und ifm_queue_next=1. Der Reihe nach stellt der round robin-Algorithmus leere Queues fest. Bei i=2 wird ifm_queue_next=0, bei i=3 wird ifm_queue_next=1. Der Ausgangswert des Indizierers ist somit wieder erreicht (vergleiche mit Bild 86). Jetzt sollte die for-Schleife auf ihre natürliche Weise abbrechen, und nicht wie bisher zwangsweise durch break. Sie tut es bei i=4; denn bei diesem Wert ist erstmalig die Bedingung i<ifm_queue_max nicht mehr erfüllt.

Fazit: Die for-Schleife garantiert, daß alle ifm_queue-Köpfe durchsucht werden, und ifm_queue_next garantiert, daß dies reihum geschieht.

Bleibt noch folgendes zu bemerken: Wenn der round robin-Algorithmus nach der Durchsuchung aller ifm_queue-Köpfe nur leere Queues festgestellt hat, setzt er zur späteren Auswertung send_pending[6] = false.
Der gefundene Algorithmus scheint ein leistungsfähiges Instrument zu sein, um alle Zielrechner (dhosts) in gerechter Weise mit potentiellen LLC PDUS zu versorgen. Aber ein kleines Problem steht noch an. Was geschieht, wenn die Variable ifm_queue_next auf ihrem Weg durch die ifm_queue-Köpfe den Wert von **host_id** annimmt (ifm_queue_next == host_id)? Antwort: Der Algorithmus ermittelt seinen eigenen Host als Zielrechner. Aber mit den folgenden Anweisungen ist dieses Problem leicht zu lösen:
if(ifm_queue_next == host_id) ifm_queue_next++ ; Wenn die Variable ifm_queue_next den Kopf des eigenen host indiziert, macht der Algorithmus beim nachfolgenden Kopf weiter, und schließt auf diese Weise den eigenen Host als Zielrechner aus.

Hier nun das komplette Programm des round robin-Algorithmus. Siehe Bild 88.

```
#define true                    1
#define ifm_queue_max     4
#define max_access_class   6

extern unsigned char      host_id;
static unsigned char      ifm_queue_next=0,i;

void ifm_round_robin_read(unsigned char *send_pending,unsigned char mac_pdu[])
{
 for(i=0;i<ifm_queue_max;i++)
   {
    if(ifm_queue_next == host_id)
      ifm_queue_next++;

    if(ifm_queue_next == ifm_queue_max)
      ifm_queue_next = 0;

    switch(ifm_queue_next)
    {
     case 0: ifm0_read(send_pending,mac_pdu);
     break;
     case 1: ifm1_read(send_pending,mac_pdu);
     break;
     case 2: ifm2_read(send_pending,mac_pdu);
     break;
     case 3: ifm3_read(send_pending,mac_pdu);
     break;
    }

    ifm_queue_next++;

    if(*send_pending == true)
      break;
   }
}
```

Bild 88: Betriebssystem-Funktion ifm_round_robin_read (Datei ifmread.c)

Die Funktion wird mit zwei Parametern aufgerufen, der Adresse &send_pending[max_access_class] des 6ten Elements aus dem array send_pending[] und der Adresse des arrays mac_pdu[]. Sie führt die beschriebene for-Schleife aus, und stellt dort am Anfang fest, ob sie den eigenen Host als Zielrechner ermittelt hat. Danach, ob sie das array ifm_queue_kopf[] zu einem Ring schließen muß. Anschließend ermittelt sie mit

switch(ifm_queue_next)

den aktuellen llc_queue_kopf[] und stellt mit ifm0_read(..) **oder** ifm1_read(..) **oder** ... ifm3_read() fest, ob im betreffenden llc_queue_kopf[] eine llc_pdu eingekettet ist (siehe Bilder 77, 78, 79, 80).

Wenn **ja**, beendet die Funktion die for-Schleife mit **break**.
Wenn **nein**, setzt die Funktion die for-Schleife fort, bis sie auf natürliche Weise bei i==ifm_queue_max endet.

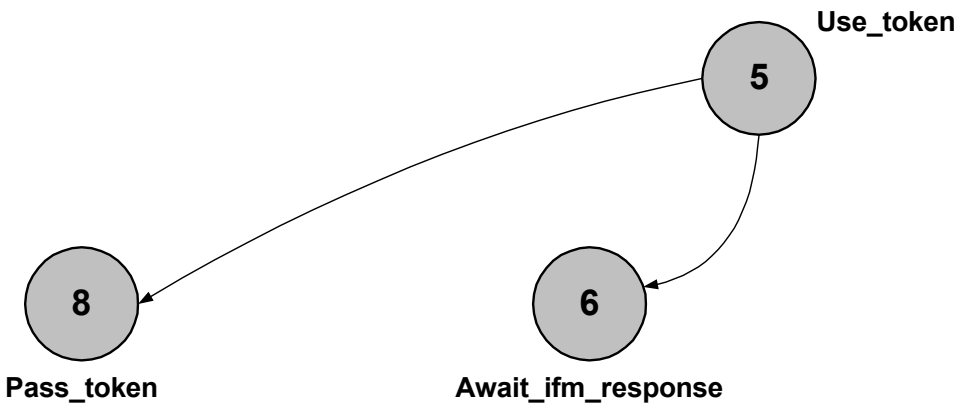
Was immer auch geschieht, nach jedem Schleifendurchgang wird ifm_queue_next inkrementiert.

Nachdem wir einen Algorithmus gefunden haben, der nach einer gerechten Strategie LLC PDUs für die Übertragung „aufspüren“ kann, wollen wir uns nun die resultierenden Aktionen der Access Control Machine im Zustand Use_Token ansehen. Betrachten wir dazu den folgenden Programm-Abschnitt:

Anweisungen

```
case use_token:

      uart_in_sende_modus();
      ifm_round_robin_read(&send_pending[max_acces_class],mac_pdu);

      if(send_pending[max_access_class] && token_hold_timer > 0)
        {
         get_tht(&token_hold_timer);
         mac_pdu[ts]    = this_station;
         mac_pdu[ctrl] = data;
         txm_exe(mac_pdu);

         acm_status   = await_ifm_response;
        }
      else
        acm_status   =  pass_token;

break;
```

Transitionen



Da mit dem Wechsel in den Zustand Use_token die Access Control Machine die Sendeberechtigung erhalten hat, schaltet sie als erstes den UART in den **Sendemodus**. Anschließend ruft sie die Funktion ifm_round_robin_read(..); auf und macht sich auf die Suche nach LLC PDUs. Hat sie eine gefunden, geht sie in den Zustand Await_ifm_response, kehrt von dort wieder zurück in den Zustand Use_Token und beginnt die Suche von neuem. Ohne besondere Maßnahmen würde dies unendlich oft geschehen. Die ACM behielte für immer das Token und bliebe somit auf ewig sendeberechtigt. Dies kann nur verhindert werden, wenn die Durchsuchung der Interface Machine-Sendeeinheit zeitlich begrenzt wird. Hier hilft der token hold timer. Er ist als Zähler implementiert und erhält im Zustand Idle oder Claim_token von set_thtr(.) den anfänglichen Wert hi_pri_token_hold_time.
Im Zustand Use_token wird mit der folgenden if-Abfrage der aktuelle Zustand des token hold timers überprüft und gleichzeitig auch der Wert von send_pending[6] festgestellt.

```
if(send_pending[max_access_class] && token_hold_timer > 0)
  {
    - - - - -
  }
```

Dabei führen die Überprüfungen zu folgenden Ergebnissen:

1, Ist das Zeitkontingent noch nicht ausgeschöpft, also token_hold_timer>0 und gleichzeitig send_pending[6]==**true** behält die Access Control Machine das Token.

2, Ist der token_hold_timer>0 und gleichzeitig send_pending[6]==**false** gibt die Access Control Machine das Token weiter an die Nachfolge-Station.

3, Ist der token_hold_timer==0, gibt die Access Control Machine das Token ebenfalls ab; send_pending[6] ist dann ohne Bedeutung.

Fazit: Die Access Control Machine behält die Sendeberechtigung nur solange, wie innerhalb der Token-Haltezeit LLC PDUs bereitstehen.

Die in der if-Abfrage genannte Variable **token_hold_timer** ist eine **boolsche** Variable, die den Zustand des token hold timers meldet. Ist dessen Zählerstand größer 0, bekommt die Variable den Wert 1. Ist der Zählerstand gleich 0, bekommt sie den Wert 0. Dies erledigt die Kernelfunktion **get_tht(&token_hold_timer)**, die zusammen mit set_thtr(.) den Zeit-Mechanismus des token hold timers realisiert. Mit jedem Aufruf verringert get_tht() den Zähler um **1** (beginnend bei hi_pri_token_hold_time) und gibt abhängig vom hinterlassenen Zählerstand der boolschen Variable den entsprechenden Wert. In der nächsten if-Abfrage wird dann dieser Wert abgefragt. Ist er 0, wird der if-Block verlassen und damit die Übertragungs-Sequenz beendet. Ist er 1, kann eine weitere Übertragung erfolgen.
Sind die Bedingungen für eine Übertragung erfüllt, muss nur noch das MAC Control-Feld „aufgefüllt" werden. So bekommt das Element mac_pdu[ts] die Nummer der Sende-Station **this_station**. Damit kann die Empfangs-Station die Quell-Station identifizieren. Und das Element mac_pdu[ctrl] bekommt die Control-Information **data**. Damit kann die Empfangs-Station die empfangene MAC PDU als Daten-Frame identifizieren. Keine Information bekommt das Element mac_pdu[ns] (ns = next station), denn die Access Control Machine kann die Nummer der Ziel-**Station** nicht wissen. Diese Nummer darf nicht mit der Nummer des Ziel-**Host** verwechselt werden; diese ist nur dem LLC Layer bekannt. Und so erinnern wir uns: Die Access Control Machine der Empfangs-Station ermittelt sich ihre Stations-Nummer aus rxm_puffer[dhost] selbst. Siehe Bild 50: Betriebssystem-Funktion sende_acm_event.
Da jetzt auch das MAC Control-Feld initialisiert ist, steht ein kompletter Daten-Frame zur Übertragung bereit. Für die Übertragung sorgt wie üblich die Transmit Machine und die ACM aktiviert sie mit **txm_exe(mac_pdu);**
Wenn eine Station einen Daten-Frame an eine andere Station sendet, dann ist für die Übertragung eine bestimmte Zeit erforderlich. Diese Zeit ist uns als Framezeit t_f bekannt und liegt in unserem Modell-Netz bei 14,6ms. Während dieser Zeit behält die ACM der Sendestation das Token. Bei der ersten Übertragung verbraucht die Sendestation die Zeit $1 \bullet t_f = 1 \bullet 14.6ms$, bei der nächsten Übertragung die Zeit $2 \bullet t_f = 2 \bullet 14.6ms$ usw.
Wir erkennen, dass die Anzahl der Übertragungen die Länge der Token-Haltezeit bestimmt. Ist die maximale Anzahl = hi_pri_token_hold_time, dann liegt sie bei

$$Token\text{-}Haltezeit = hi\_pri\_token\_hold\_time \bullet t_f$$

Hat die Transmit Machine einen Daten-Frame übertragen, geht die Access Control Machine in den Zustand **Await_ifm_response** und gibt der Ziel-Station Gelegenheit auf den empfangenen Daten-Frame mit einem sogenannten **Response**-Frame zu antworten. In diesem Zustand wartet die ACMTask nur eine begrenzte Zeit auf die Ankunft dieses Frame. Sie liegt bei 2•slot_time, denn der Daten-Frame braucht 1•slot_time, um zur Zielstation zu gelangen und der Response Frame braucht die gleiche Zeit, um zur Quellstation zurückzukommen. Da während dieser Aktion die ACM der Quellstation keinen Daten-Frame sendet, wäre es Unsinn, den token hold timer weiterlaufen zu lassen.
Konnte ifm_round_robin_read(..); während der Token-Haltezeit **keine** LLC PDU feststellen, oder ist der token hold timer abgelaufen (token_hold_timer == 0), dann hat die Station keine Sendeberechtigung mehr. Die Access Control Machine gibt deshalb das Token weiter an ihre Nachfolge-Station.
Und weil dies im Zustand **Pass_token** geschieht, veranlasst sie mit acm_status = pass_token den notwendigen Zustandswechsel.

1.3.2.1.4 Der Zustand Pass_token

In einem tokenbus-basierten LAN ist jeder Station ts (this_station) die Vorgängerstation ps (previous station) und die Nachfolgestation ns (next_station) bekannt. Ihre Kennummern werden in der Initialisierungsphase von der (noch nicht besprochenen) Betriebssystem-Funktion **create_acm(..);** festgelegt. Für die Access Control Machine im Zustand Pass_token sind in der folgenden Implementierung nur die Kennummern der eigenen Station this_station und der Nachfolgestation next_station von Interesse. Und so sieht die Übertragung des Token an einem Beispiel wie folgt aus. Siehe Bild 89.

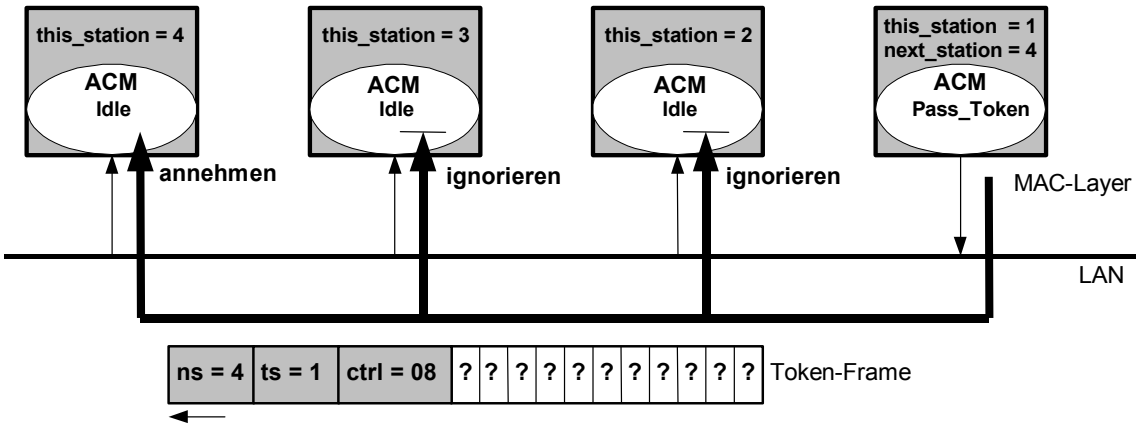


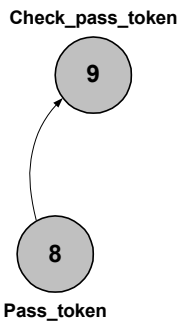
Bild 89: Token-Weitergabe

Die Station 1 (this_station=1) war im Zustand Use_token und befindet sich jetzt im Zustand Pass_token. In diesem Zustand gibt sie das Token weiter an ihre Nachfolgestation (next_station=4). Damit dies geschehen kann, lädt die Access Control Machine zunächst die Variablen next_station und this_station in die MAC-Elemente mac_pdu[ns] und mac_pdu[ts]. Das MAC-Element mac_pdu[ctrl] bekomt die Control-Information 0x08. Die restlichen Elemente bleiben undefiniert. So ist der Token-Frame komplett, und die Access Control Machine leitet ihn mit txm_exe(mac_pdu) ; zur Übertragung weiter an die Transmit Machine. Die Anweisungen der Access Control Machine im Zustand Pass_Token gestalten sich danach wie folgt :

Anweisungen

```
case pass_token:

    uart_in_sende_modus();
    mac_pdu[ns]     = next_station;
    mac_pdu[ts]     = this_station;
    mac_pdu[ctrl]   = token;
    txm_exe(mac_pdu);

    acm_status      = check_pass_token;

break;
```

Transition



1.3.2.1.5 Der Zustand Check_pass_token

Hat die Access Control Machine das Token an die Nachfolgestation weitergeleitet, will sie nur eines « wissen » : Ist das Token wohlbehalten angekommen ? Eine Station, die das Token empfangen hat, geht vom Zustand Idle über in den Zustand Use_Token und beginnt zu senden.
Stehen LLC PDUs bereit, schickt sie einen Daten-Frame ins LAN, wenn nicht, einen Token-Frame. Wie kann die Station, die das Token abgegeben hat, diese Aktivitäten feststellen ? Sie geht vom Zustand Pass_token über in den Zustand Check_pass_token, schaltet dort als erstes den UART in den Empfangsmodus und bringt danach mit **empfange_acm_event(....)** ; die ACMTask in den Wartezustand. Die Task verweilt dort eine begrenzte Zeit und « hofft » auf das Eintreffen einer MAC PDU von der Token-Empfangsstation. Siehe dazu das folgende Programm-Fragment:

```
case check_pass_token:

    uart_in_empfangs_modus();

    token_pass_timer  = 2*slot_time;
    empfange_acm_event(token_pass_timer,&acm_event,&frame_control,
                       &dest_station,&source_station,&source_host);

    switch(acm_event)
    {
       ------
    }

break;
```

Trifft ein Ereignis ein, wird die Task aus dem Wartezustand befreit, und die Access Control Machine setzt ihre Arbeit fort. Sie stellt fest, um welches Ereignis es sich handelt und überprüft zu diesem Zweck die empfangene Variable **acm_event**. Diese kann drei verschiedene Botschaften enthalten:

- timeout (-2),
- io_error (-1) und
- ok (0)

Untersuchen wir zunächst die Botschaft timeout. Der Access Control Machine ist „bewußt“, daß ein tokenbus-basiertes LAN ein „lebendiges“ System ist. So können während des Betriebs Stationen aus dem Ring entfernt oder in den Ring aufgenommen werden. Eine starre Reihenfolge der Stationen, gekennzeichnet durch fortlaufende Kennummern, ist oft nicht gegeben. Sie liegt nur dann vor, wenn sich **alle** Stationen im Ring aufhalten. Ist **n** die Kennummer der ersten Station (first_station), dann lautet die Reihenfolge:

n, n-1, n-2,.....n-(n-1).

Die Nummer der letzten Station ist also immer 1. Beispiel: Ist n=4, lautet die Reihenfolge 4, 3, 2, 1. Die Kennummer 0 kommt, wie bereits früher erläutert, **nicht** vor (siehe Bild 50: Betriebssystem-Funktion sende_acm_event).

- Deshalb lautet dann die Reihenfolge im Ring: 4, 3, 2, 1 --> 4, 3, 2, 1 --> 4, 3, 2, 1 usw.

Betrachten wir nun einige Szenarien. Siehe als erstes Bild 90.

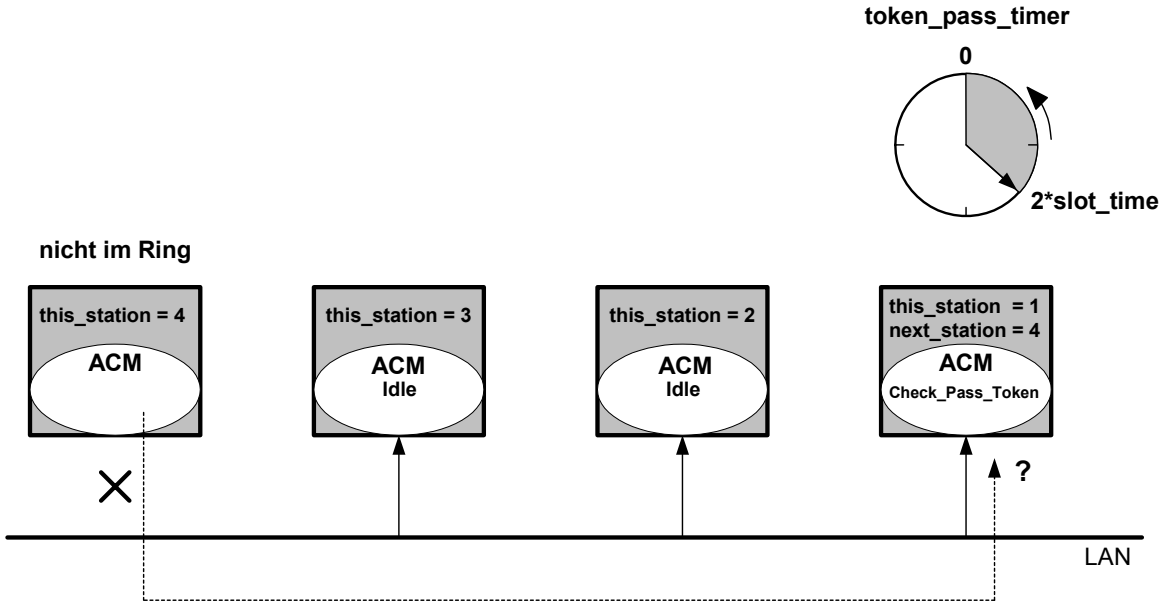


Bild 90: Erfolglose Token-Weitergabe

Angenommen, Station 1 (this_station=1) war im Zustand Pass_token und hat ihrer Nachfolgestation next_station=4 das Token überreicht. Station 1 geht in den Zustand Check_pass_token und wartet auf eine Reaktion von Station 4. Unglücklicherweise ist diese Station nicht im Ring. Sie kann das Token nicht empfangen, und natürlich auch nicht reagieren. Die Station 1 würde vergebens auf eine MAC PDU von Station 4 warten, wäre da nicht der **token_pass_timer,** den die Funktion empfange_acm_event(....); auf die maximale Wartezeit 2*slot_time setzt. So läuft der timer ab und bei 0 schickt die Zeitverwaltung des Betriebssystemkerns der ACMTask die Botschaft **acm_event=timeout**.
Wie reagiert die Access Control Machine auf diese Botschaft? Sie „weiß“, das Token ist nicht angekommen. Sie kann das Token noch einige male schicken (nach IEEE 802.4 bis zu 6 mal), oder sofort an die nächste Station weitergeben. Im letzteren Fall dekrementiert sie die Variable next_station, geht in den Zustand Pass_token und überträgt das Token jetzt an die neue Nachfolgestation (next_station=3). Danach überprüft sie im Zustand Check_pass_token die Aktivitäten dieser Station. Wenn auch sie nicht reagiert (timeout), beginnt die gleiche Prozedur mit der darauffolgenden Station (next_station=2), bis im Extremfall next_station = this_station ist. Die originale Token-Sendestation ermittelt sich selbst als Nachfolgestation. Siehe Bild 91:

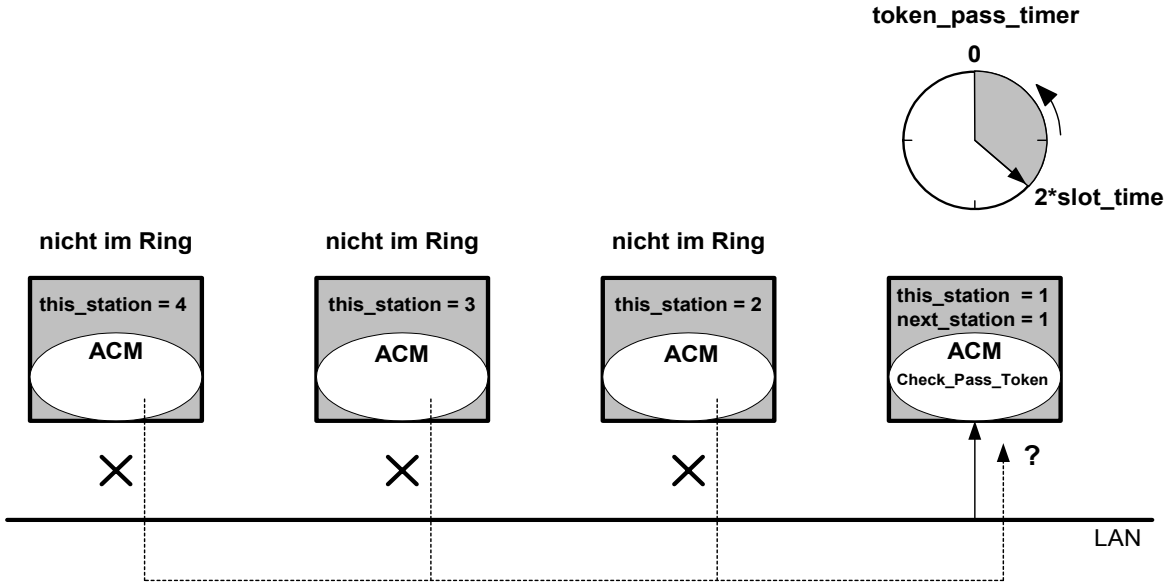


Bild 91: Station 1 ermittelt sich selbst als Nachfolgestation

Dies ist natürlich unakzeptabel. Durch nochmaliges Dekrementieren von next_station könnte eine potentielle Nachfolgestation ermittelt werden, doch die Variable bekäme den Wert 0. Eine Stationsnummer, die es im tokenbus-basierten LAN nicht gibt. Und so löst die Access Control Machine das Problem mit der zusätzlichen Variablen **next_init**. Sie wird wie die Variable next_station während der „Hochlaufphase“ des Rechners von der bereits erwähnten Betriebssystem-Funktion create_acm(..); mit der gleichen anfänglichen Kennummer der Nachfolgestation initialisiert.

Beispiele: this_station=2 -> next_station=1 -> next_init=1
this_station=1 -> next_station=4 -> next_init=4 usw.

Während sich der Wert von next_station ständig verändern kann, bleibt der Wert von next_init erhalten. Hat nach next_station--; die Variable den Wert von this_station erreicht, bekommt sie den Wert von next_init. Im Beispiel nach Bild 91 den Wert 4. Somit lauten die Anweisungen allgemein wie folgt:

```
switch(acm_event)
{
 case timeout:
       acm_status = pass_token;
       next_station--;
          - - - - -
          - - - - -
          if(next_station == this_station)
             next_station =   next_init;
 break;
    - - - -
    - - - -
}
```

Für das beschriebene Problem ist eine Lösung gefunden, doch ein weiteres „steht schon vor der Tür“. Betrachten wir dazu die folgende Situation im Bild 92.

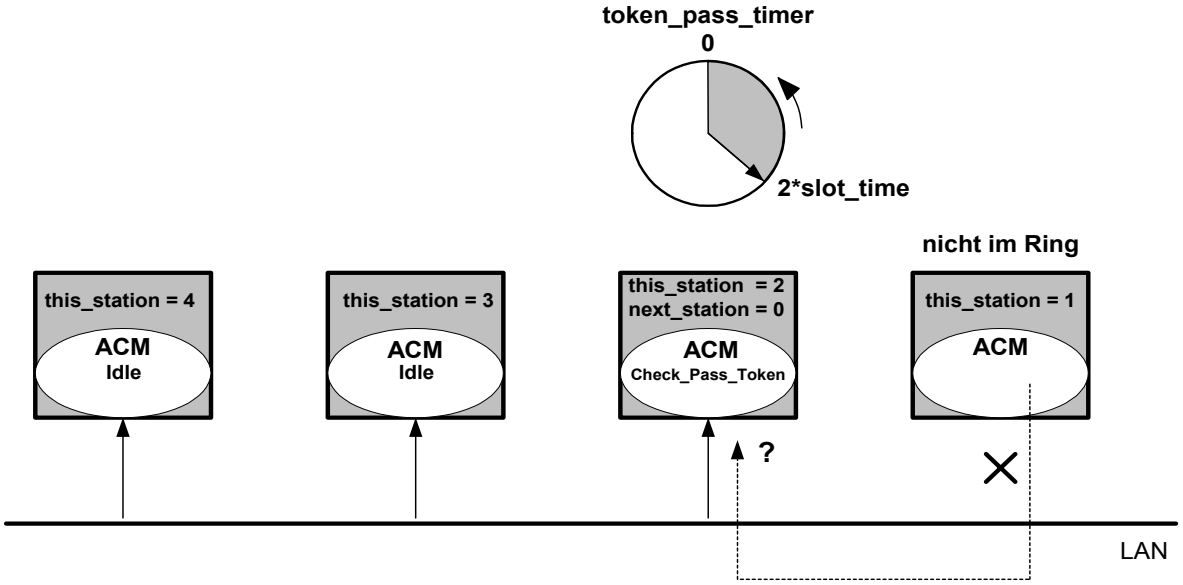


Bild 92: Nachfolgestation im Ring ermitteln

Station 2 (this_station=2) war im Zustand Pass_token und hat an ihre Nachfolgestation next_station=1 das Token weitergegeben. Station 2 ist jetzt im Zustand Check_pass_token und wartet auf eine Reaktion von Station 1. Doch diese Station ist nicht im Ring. So erhält Station 2 vom Betriebssystemkern die Botschaft acm_event=timeout. Sie muß eine neue Nachfolgestation ermitteln und dekrementiert deshalb die Variable next_station. Diese bekommt den Wert 0. Was ist zu tun? Wenn es zu dieser Situation kommt, muß die Access Control Machine den Token Bus zu einen Ring schließen und ab der ersten Station „weitersuchen“. Sie realisiert dies mit Hilfe der neuen Variablen **first_station**. Genauso wie die Variablen this_station, next_station und next_init, wird auch sie während der „Hochlaufphase“ des Rechners von der Betriebssystem-Funktion creat_acm(..); initialisiert. Ihr Wert bleibt während des Betriebs unveränderlich erhalten und richtet sich nach der Anzahl der Stationen im LAN. Enthält das LAN beispielsweise 4 Stationen, bekommt die Variable in allen Stationen den Wert 4 (first_station=4). So kann jetzt mit Hilfe von first_station der Ring auf einfache Weise mit den folgenden Anweisungen geschlossen werden:

- **if(next_station == 0)**
  next_station = first_station;

Die Frage ist, an welche Programmposition im case timeout-Block müssen diese Anweisungen platziert werden? Betrachten wir dazu eine letzte Situation im Bild 93.

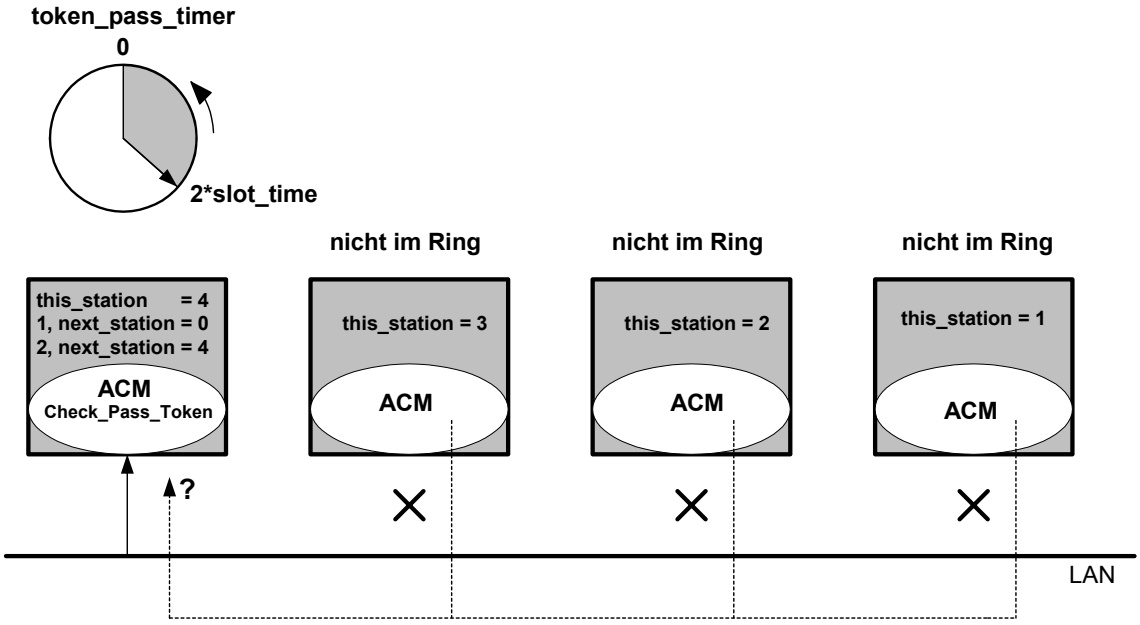


Bild 93: Station 4 schließt den Ring und ermittelt sich selbst als Nachfolgestation

Die erste Station 4 (this_station=4) ist alleine im LAN, während sich alle anderen außerhalb befinden. Station 4 gibt der Reihe nach das Token weiter an die Stationen 3, 2 und 1, und da auch die letzte Station 1 nicht antwortet, bekommt next_station den Wert 0. Die Access Control Machine von Station 4 schließt daraufhin den Ring und initialisiert gemäß den obigen Anweisungen next_station mit first_station. Doch damit ermittelt sie sich selbst als Nachfolgestation (this_station=4; first_station=4). Und so bleibt nur eines, nämlich unmittelbar danach next_station mit next_init zu laden. So schließt sich die Station selbst aus, und sie geht anschließend von neuem auf die „Suche“ nach einer Nachfolgestation. Denn zwischenzeitlich könnten ja weitere Station zugeschaltet worden sein.
Damit haben wir die Antwort auf die obige Frage gefunden. Siehe die folgenden Anweisungen: Zuerst überprüft die Access Control Machine, ob der Ring geschlossen werden muß und danach, ob sie sich selbst als Nachfolgestation ermittelt hat.

```
switch(acm_event)
{
 case timeout:
        acm_status = pass_token;
        next_station--;

        if(next_station == 0)
           next_station =   first_station;

          if(next_station == this_station)
             next_station =   next_init;
 break;
    - - - -
    - - - -
}
```

Betrachten wir nun den Fall der **erfolgreichen** Token-Weitergabe. Siehe dazu die folgende Situation im Bild 94.

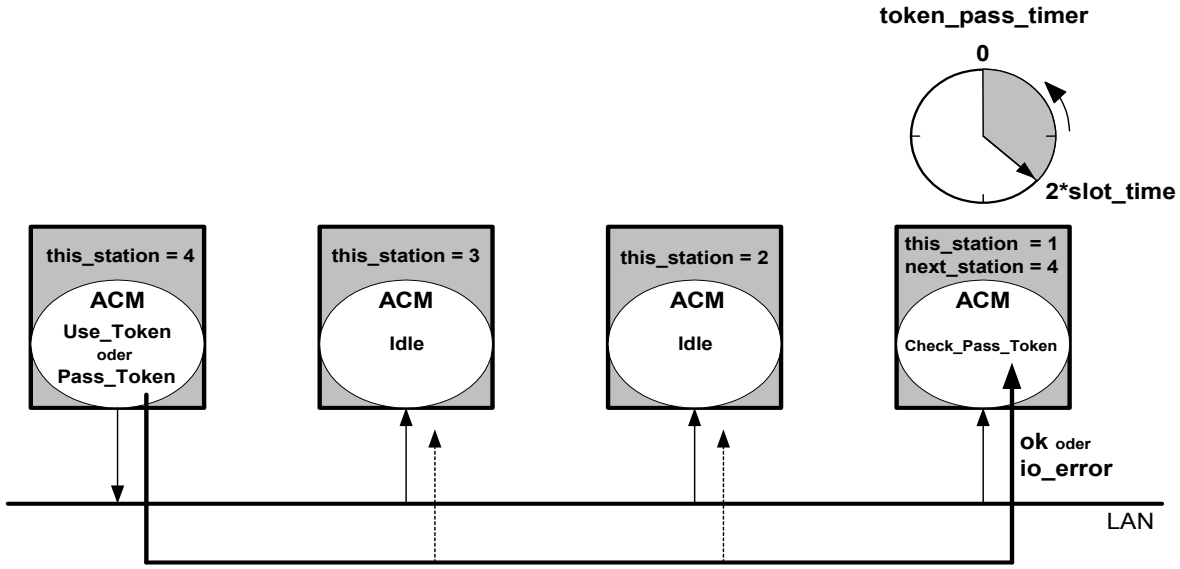


Bild 94: Erfolgreiche Token-Weitergabe

Station 1 (this_station=1) war im Zustand Pass_token und hat ihrer Nachfolgestation next_station=4 das Token überreicht. Station 1 geht in den Zustand Check_pass_token und wartet auf eine Reaktion von Station 4. Sie ist glücklicherweise im Ring, und kann somit das Token empfangen und darauf reagieren.
Die Access Control Machine von Station 4 sucht im round robin-Verfahren innerhalb der token_hold_time in ihren ifm_queue-Köpfen nach eingeketteten LLC PDUs. Sie „packt“ die erste, die sie findet, in eine MAC PDU und schickt sie als Daten-Frame ins LAN. Dies erledigt sie im Zustand Use_token.
Konnte die Access Control Machine keine LLC PDU finden, gibt sie das Token weiter und schickt einen Token-Frame ins LAN. Dies erledigt sie im Zustand Pass_token.
Die Stationen 3 und 2 sind im Idle-Zustand und die Station 1 ist im Zustand Check_pass_token.
Alle diese Stationen „hören“ daher am LAN mit und empfangen entweder einen Daten-Frame oder einen Token-Frame. Die Station 1 hat kein Interesse daran, um welchen Frame es sich handelt und für wen der Frame bestimmt ist. Sie bekommt die Botschaft **acm_event=ok**. Damit hat für sie die Station 4 reagiert, und die Token-Weitergabe ist für sie erfolgreich abgeschlossen. Sie kann sich jetzt „zurücklehnen“, also den Zustand Check_pass_token verlassen und in den Zustand Idle übergehen.
Wie verhält sich Station 1, wenn die MAC PDU von Station 4 beschädigt bei ihr ankommt? Sie empfängt dann die Botschaft **acm_event=io_error**. Station 1 steht auf dem Standpunkt, daß Station 4 reagiert hat, lediglich der Bitstrom ist zerstört worden. Und so interpretiert sie die Botschaft als „Token-Weitergabe wahrscheinlich ok“, verläßt auch hier den Zustand Check_pass_token und geht über in den Zustand Idle.

Da sich die Access Control Machine im Zustand Check_pass_token beim Empfang der Ereignisse ok oder io_error identisch verhält, müssen sie im switch-Block nicht explizit festgestellt und ausgewertet werden. Dies kann alleine unter default geschehen. Somit lauten die Anweisungen:

```
switch(am_event)
 {
   case timeout:
         - - - -
         - - - -
    break;

    default:                                /* ok oder io_error */
         acm_status   = idle;
         next_station = next_init;
    break;
   }
```

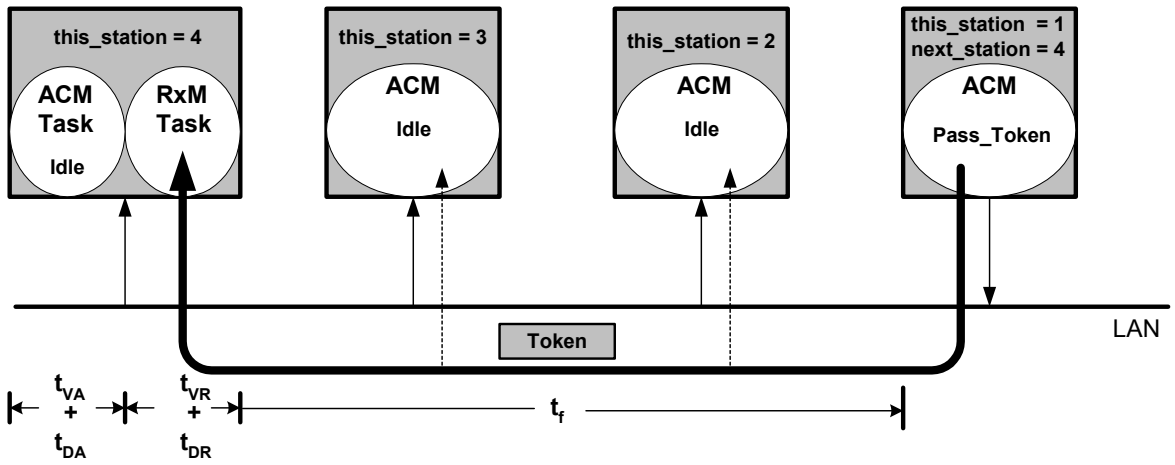
Sollte das Token an eine Station weitergegeben worden sein, die nicht die natürliche Nachfolgerin der Token-Sendestation ist, dann ist hier außerdem ein guter Platz, um die ursprüngliche Ordnung wieder herzustellen. Was damit gemeint ist, soll das folgende Beispiel verdeutlichen:

Beispiel (vergleiche mit Bild 94): Die Station 4 sei **nicht** im Ring. Station 1 gibt daher im Zustand Pass_token das Token weiter an Station 3. Im Zustand Check_pass_token stellt Station 1 die Weitergabe als erfolgreich fest und geht in den Zustand Idle. Während die Station 3 und danach Station 2 senden, könnte Station 4 als natürliche Nachfolgestation von Station 1 in den Ring aufgenommen worden sein. Kommt das Token irgendwann bei Station 1 an, kann sie jetzt nach Erledigung ihres Sendeauftrags die Token-Weitergabe wegen **next_station=next_init** wieder bei ihrer „normalen“ Nachfolgestation beginnen.

Zusammengefasst gestalten sich die Anweisungen der Access Control Machine im Zustand Check_pass_token wie folgt:

```
case check_pass_token:
      uart_in_empfangs_modus();

      token_pass_timer  = 2*slot_time;
      empfange_acm_event(token_pass_timer,&acm_event,&frame_control,
                         &dest_station,&source_station,&source_host);

      switch(acm_event)
      {
       case timeout:                              /* pass_token_failed */
            acm_status = pass_token;
            next_station--;

            if(next_station == 0)
               next_station =  first_station;

            if(next_station == this_station)
               next_station =  next_init;
       break;

       default:                                    /*  ok oder io_error  */
            acm_status   = idle;                   /*        pass ok        */
            next_station = next_init;
       break;
      }
break;
```

Nachdem wir nun die logischen Mechanismen im Check_pass_token-Zustand kennengelernt haben, wollen wir abschließend auch das zeitliche Verhalten dieses Zustands untersuchen. Die Frage lautet: Welche Kriterien bestimmen die Länge der Wartezeit? Betrachten wir dazu die folgenden Korrespondenzen in den Bildern 95a und 95b.



t_f: Framezeit
t_{VR}: Verzögerungszeit der RxMTask
t_{DR}: Dienstzeit der RxMTask
t_{VA}: Verzögerungszeit der ACMTask
t_{DA}: Dienstzeit der ACMTask

Bild 95a: Station 1 sendet Token an Station 4

Station 1 ist im Zustand Pass_token und gibt das Token weiter an ihre Nachfolgestation 4. Der Token-Frame „besetzt" das LAN für die Dauer der Framezeit t_f, und der UART von Station 4 nimmt ihn Byte für Byte in seinen FIFO auf. Beim letzten Byte löst der UART einen Interrupt aus und aktiviert den Interrupt-Handler ir29_handler (siehe Bild 61). Dieser ruft die Betriebssystem-Funktion signal_ir29() auf, die nun ihrerseits der wartenden RxMTask das Eintreffen einer MAC PDU, in unserem Fall des Token-Frame, signalisiert. Daraufhin läuft folgendes ab:

- Der Task-Control-Block (TCB) der RxMTask wird an der höchsten Prioritätsstufe der Task-Queues eingekettet.
- Der Task-Control-Block der augenblicklich aktiven Task wird aus ihrer aktuellen Task-Queue ausgekettet.
- Der Scheduler wird aktiviert, und ermittelt die RxMTask als diejenige mit der augenblicklich höchsten Priortät.
- Der Dispatcher startet die RxMTask, die daraufhin ihre Arbeit beginnt.

Die Zeit, die zwischen dem Eintreffen des Ereignisses (in unserem Fall des Interrupts) und der Reaktion der RxMTask vergeht, wird als Reaktionszeit oder **Taskverzögerungszeit** bezeichnet. Wir nennen sie $\mathbf{t_{VR}}$.
Die RxMTask erledigt ihre Arbeit (fcs-Berechnung usw.) und schickt schließlich mit sende_acm_event(..); eine Botschaft an die ACMTask. Die verbrauchte Zeit zwischen dem Start der RxMTask und dem erfolgreichen Versand der Botschaft bezeichnen wir als **Taskdienstzeit**, und nennen sie $\mathbf{t_{DR}}$.

Die ACMTask war während des Ablaufs dieser beiden Zeiten im Wartezustand und wird jetzt von sende_acm_event(..); aufgeweckt. Dabei läuft folgendes ab:

- Der Task-Control-Block der ACMTask wird an ihrer aktuellen Prioritätsstufe in die Task-Queue eingekettet.
- Der Task-Control-Block der noch aktiven RxMTask wird aus ihrer Task-Queue ausgekettet.
- Der Scheduler wird aktiviert, und ermittelt die Task mit der augenblicklich höchsten Priorität. Dies muß **nicht** die ACMTask sein, denn sie bewirbt sich zusammen mit den restlichen Task des Systems (ausgenommen die RxMTask) um die Zuteilung an den Prozessor. Wird sie schließlich ausgewählt,
- startet sie der Dispatcher, und sie beginnt daraufhin ihre Arbeit.

Die Verzögerungszeit der ACMTask, wir nennen sie $\mathbf{t_{VA}}$, also die Zeit zwischen dem Eintreffen des Botschaft (acm_event) und ihre Reaktion darauf, ist nicht vorherbestimmbar. Sie ist variabel, und der relevante größte Wert kann nur meßtechnisch ermittelt werden.
Ist die ACMTask schließlich im Besitz der CPU geht ihre Access Control Machine in den Zustand Use_Token und sucht nach LLC PDUs in der Interface Machine-Sendeeinheit. Bei Erfolg schickt sie einen Daten-Frame ins LAN, bei Mißerfolg einen Token-Frame. In beiden Fällen nutzt sie die Transmit Machine. So beansprucht auch die ACMTask eine bestimme Dienstzeit, und wir nennen sie $\mathbf{t_{DA}}$.

Fazit: Erst nach Ablauf dieser 5 Zeiten macht sich die Token-Empfangsstation im LAN „bemerkbar“. Wir nennen sie Verzögerungszeit $\mathbf{t_{VSt}}$ der Station und schreiben dafür

$$\mathbf{t_{VSt} = t_f + t_{VR} + t_{DR} + t_{VA} + t_{DA}}$$

Es ist einsichtig, daß diese Zeit Einfluß auf die einzustellende Wartezeit im Zustand Check_pass_token hat. Fahren wir also fort, und untersuchen das weitere Geschehen. Siehe dazu Bild 95b.

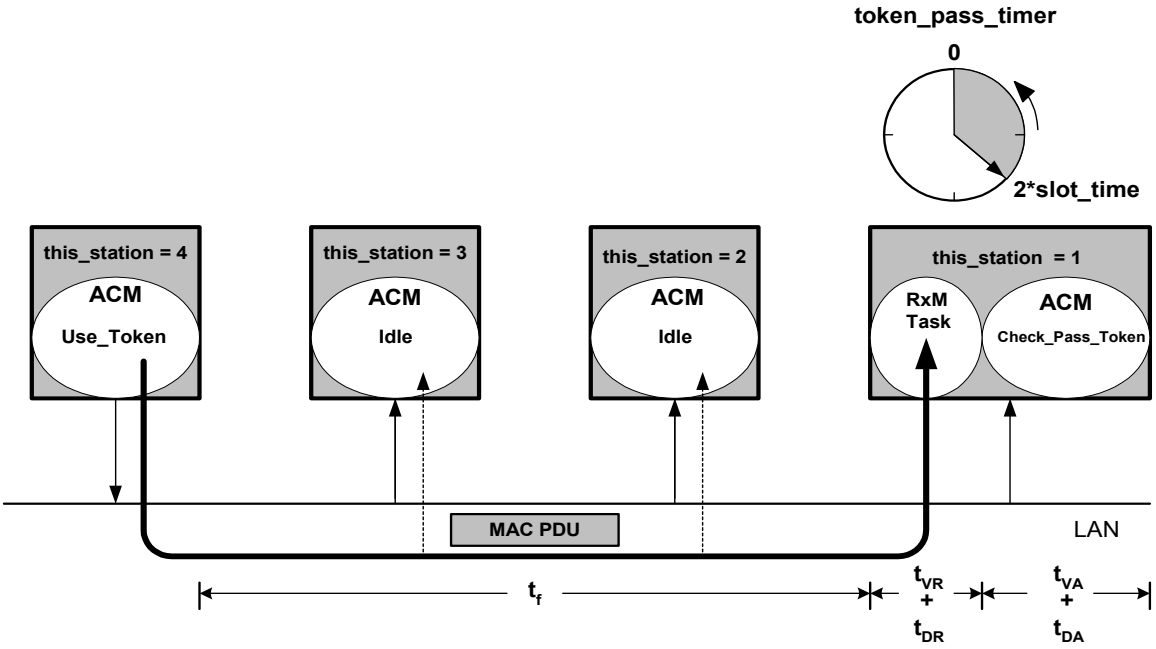


Bild 95b: Station 4 sendet MAC PDU ins LAN

Die Access Control Machine von Station 1 ist längst im Zustand Check_pass_token und wartet dort auf eine MAC PDU (einen Daten- oder Token-Frame) von Station 4. Die MAC PDU bleibt die Framezeit t_f im LAN. Bis Station 1 schließlich die Reaktion von Station 4 erkennen kann, vergeht die Verzögerungszeit t_{VR} und die Dienstzeit t_{DR} der RxMTask und es vergeht die Verzögerungszeit T_{VA} und die Dienstzeit t_{DA} der ACMTask. Wenn wir zur Vereinfachung die Verzögerungszeiten aller Stationen gleich setzen, dann gilt auch hier:

$$\mathbf{t_{VSt} = t_f + t_{VR} + t_{DR} + t_{VA} + t_{DA}}.$$

Damit sind alle Kriterien zur Bestimmung der Wartezeit im Zustand Check_pass_token bekannt. Wir definieren die Zeit **slot_time**, und geben ihr folgende Eigenschaft: slot_time $\geq t_{VSt}$.

Daraus lässt sich schließlich die **Wartezeit** ableiten, sie lautet:

$$\mathbf{2*slot\_time \geq 2*t_{VSt} = 2*(t_f + t_{VR} + t_{DR} + t_{VA} + t_{DA})}$$

Ist die Wartezeit zu klein gewählt, kann die Token-Sendestation die Reaktion der Token-Empfangsstation nicht erkennen. Die wartende Token-Sendestation bekommt die Meldung **timeout** und ihr Fehlschluss lautet: „Station nicht im Ring“. Ist die Wartezeit zu groß gewählt und die potenzielle Token-Empfangsstation tatsächlich **nicht** im Ring, vergeudet die Token-Sendestation wertvolle Zeit, bis sie sich endlich auf die Suche nach ihrer neuen Nachfolgestation macht.

Von besonderem Interesse ist dabei das Verhalten der **bus_idle_timer**. Gibt es einen Zusammenhang zwischen der Wartezeit einer ACM im Zustand Check_pass_token und den Wartezeiten der restlichen ACMs, die sich im Zustand Idle befinden?
Angenommen Station 4 ist im Besitz des Token, Station 3 ist **nicht** im Ring und die Stationen 2 und 1 befinden sich im Idle-Zustand. Die bus_idle_timer werden alle mit

bus_idle_timer = 7•slot_time geladen.

Somit warten

- Station 1 als auch Station 2 maximal 7•slot_time

Die zeitliche Abfolge der Ereignisse soll nun anhand der folgenden Bilderreihe (Bild 96a..Bild 96??) untersucht werden.

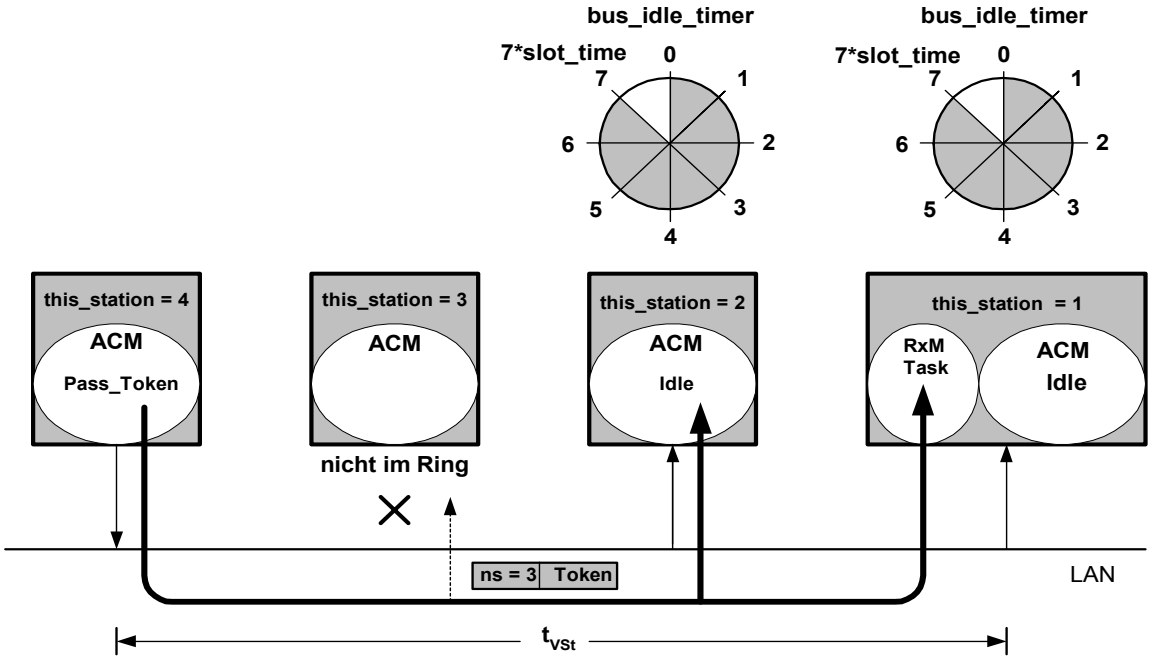


Bild 96a:

Station 4 geht in den Zustand Pass_token und schickt das Token zur Weitergabe an ihre Nachfolgestation 3 ins LAN. Das Token kann nur von Station 2 und Station 1 aufgenommen werden, aber nicht von der Zielstation 3. Bis zur Ankunft des Tokens bei den ACMs der Stationen 2 und 1 vergeht die Stationsverzögerungszeit t_{VSt} (siehe Bild 96a).

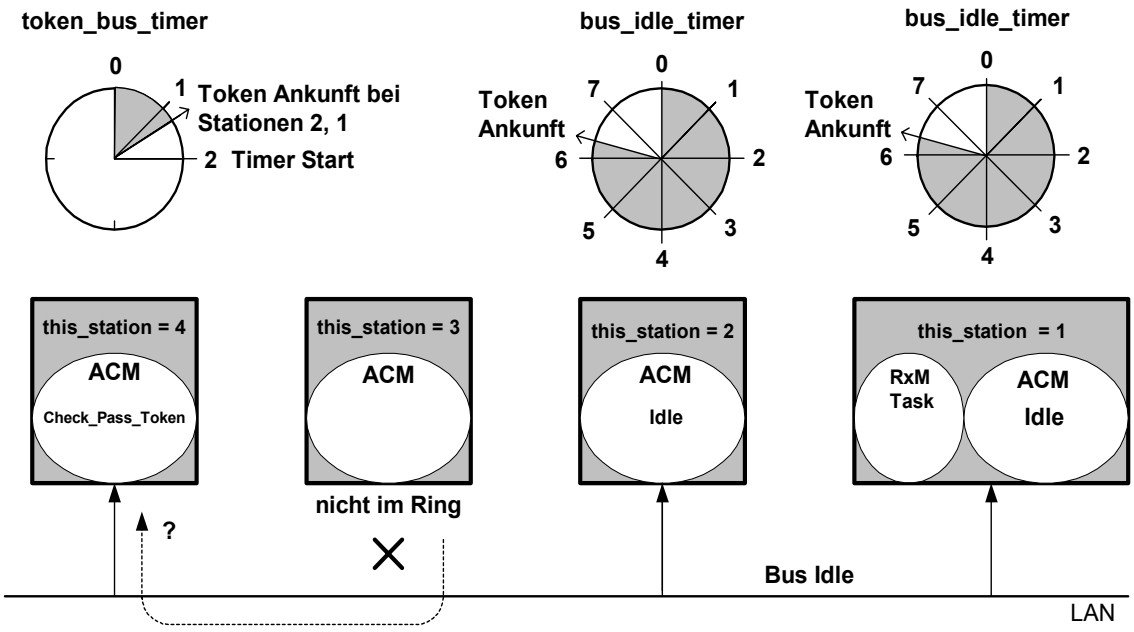


Bild 96b:

Unmittelbar nach der Ausgabe des Tokens geht die ACM der Station 4 in den Zustand Check_pass_token und der token_bus_timer wird mit der Wartezeit 2•slot_time gestartet. Alle drei timer laufen jetzt parallel ab. Während die ACM der Station 4 auf eine Reaktion von Station 3 wartet, trifft das Token bei den ACMs der Stationen 2 und 1 ein, worauf ihre bus_idle_timer stoppen (siehe Bild 96b).

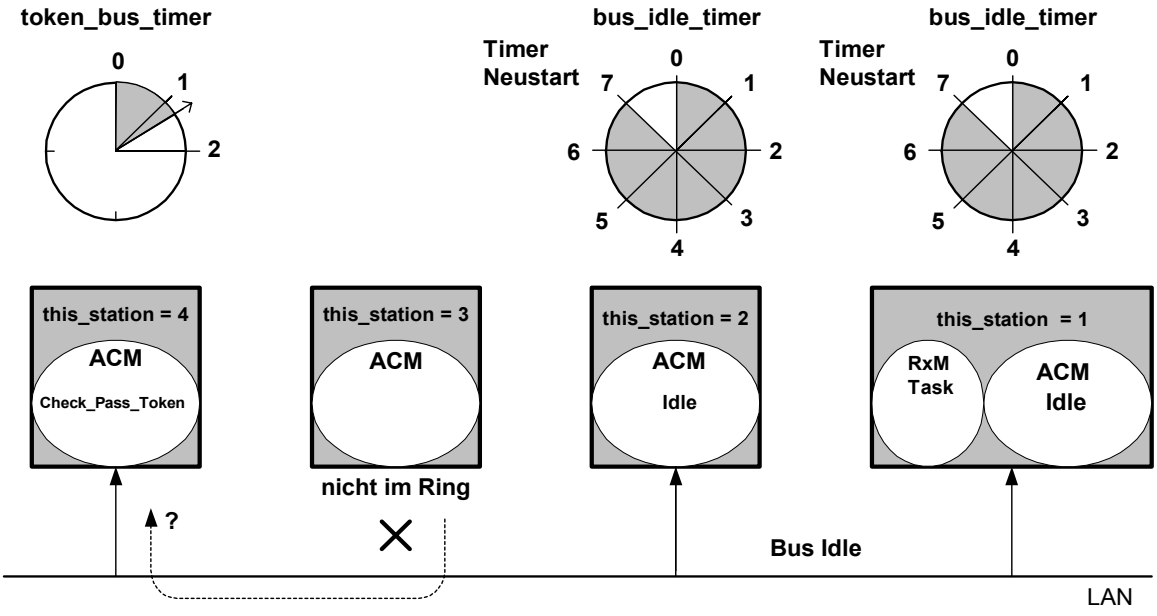


Bild 96c:

Die ACMs der Stationen 2 und 1 bleiben im Zustand Idle, denn das Token ist nicht für sie bestimmt. So werden die bus_idle_timer beider Stationen mit ihren anfänglichen Wartezeiten neu gestartet (siehe Bild 96c).

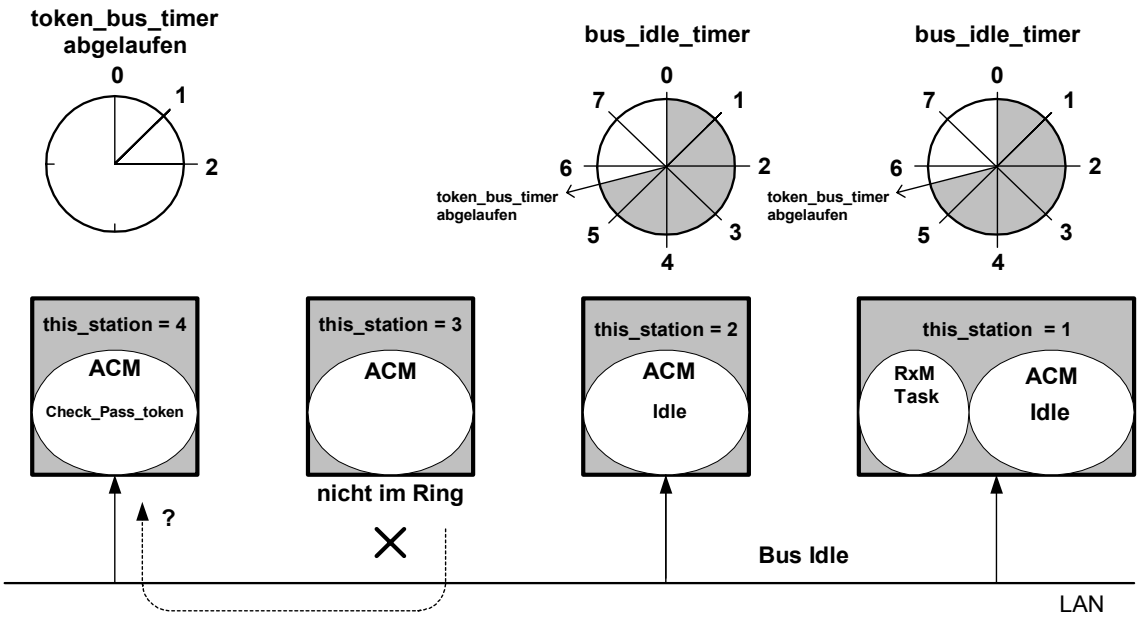


Bild 96d:

Während Station 4 weiterhin vergeblich auf eine Reaktion von Station 3 wartet, läuft der token_bus_timer ab und erreicht den Wert 0. Parallel dazu verringern sich die Wartezeiten der Stationen 2 und 1 um die gleiche vom token_bus_timer verbrauchte Restzeit (siehe Bild 96d).

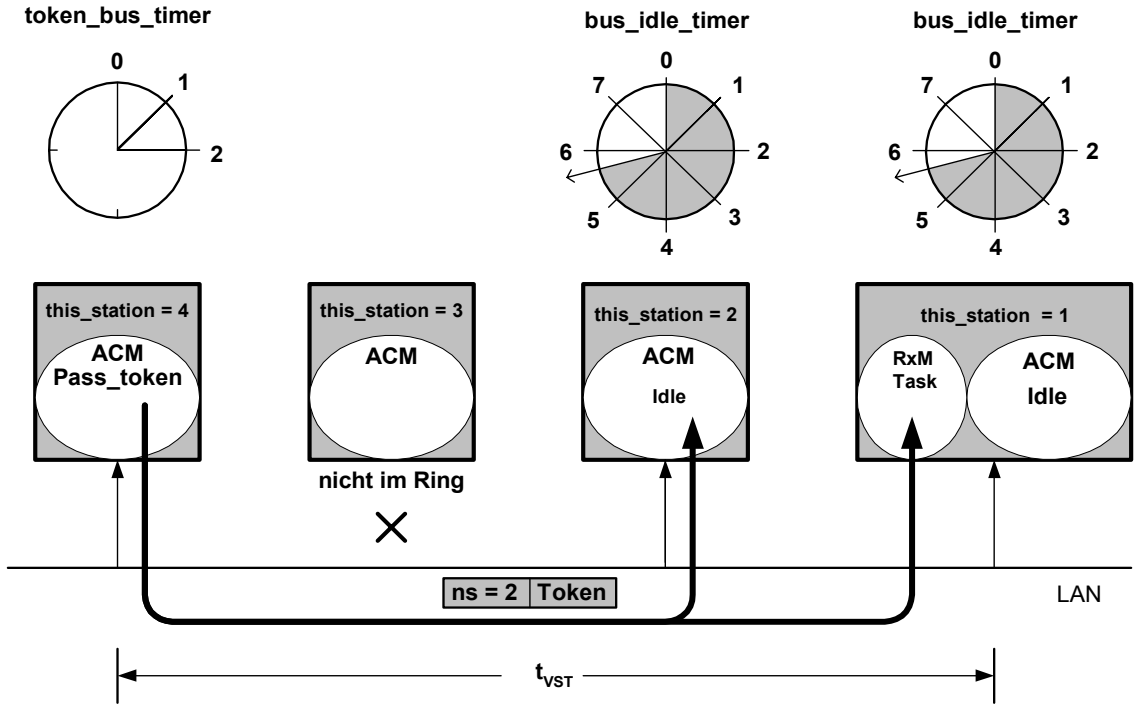


Bild 96e:

So geht die ACM der Station 4 in den Zustand Pass_token, und ermittelt als ihre neue Nachfolgestation die Station 2. Sie schickt das Token ins LAN und die Stationen 2 und 1 nehmen das Token auf. Bis das Token bei den dortigen ACMs ankommt, vergeht auch hier die Stationsverzögerungszeit $\mathbf{t_{VST}}$ (siehe Bild 96e).

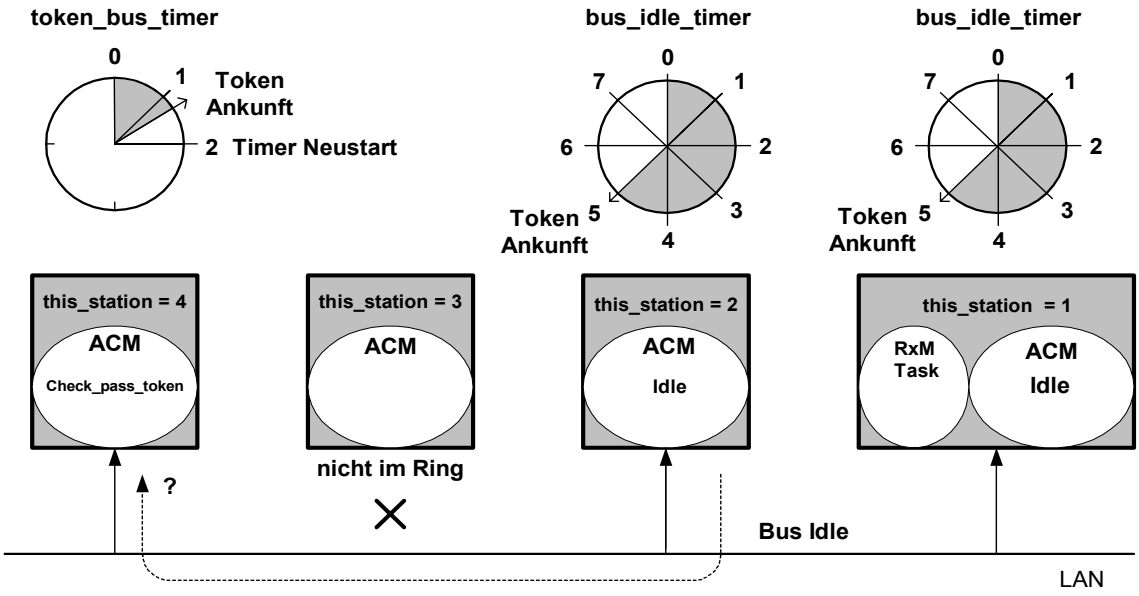


Bild 96f:

Nach Ausgabe des Tokens geht die ACM der Station 4 in den Zustand Check_pass_token und der token_bus_timer wird wieder mit der Wartezeit 2•slot_time gestartet. Alle drei timer laufen parallel ab. Während die ACM der Station 4 auf eine Reaktion von Station 2 wartet, trifft das Token bei den ACMs der Stationen 2 und 1 ein, und ihre bus_idle_timer werden gestoppt (siehe Bild 96f).

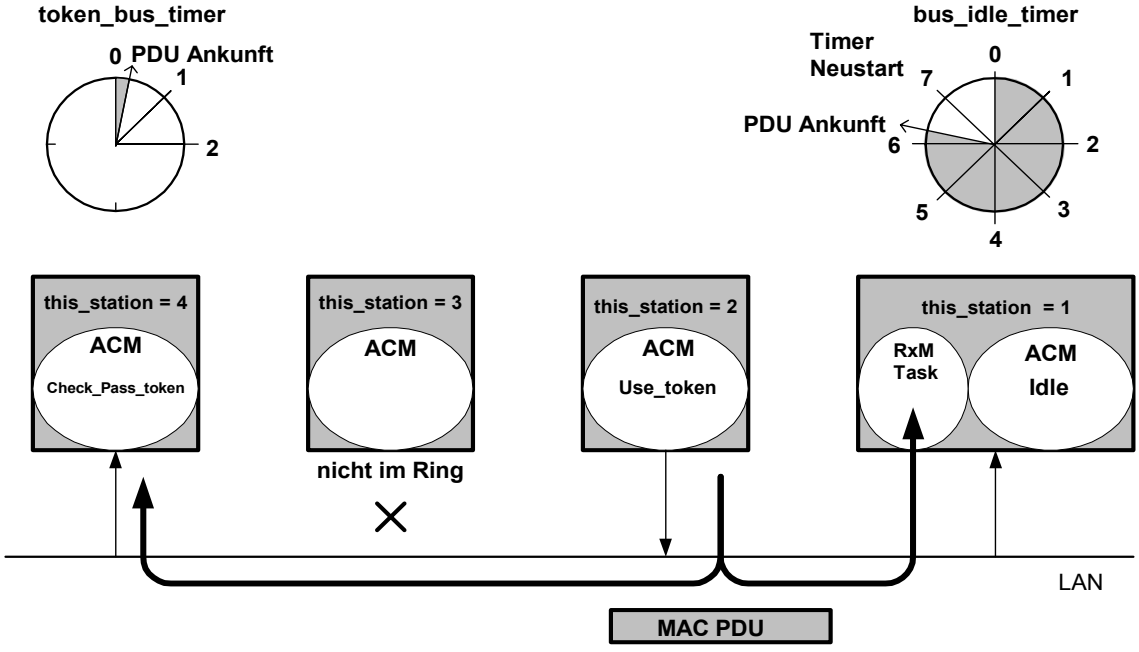


Bild 96g:

Weil das Token **nicht** für Station 1 bestimmt ist, bleibt deren ACM im Zustand Idle und ihr bus_idle_timer wird neu gestartet. Für die Station 2 ist dagegen das Token bestimmt. Ihre ACM geht in den Zustand Use_token, ermittelt eine LLC PDU und schickt sie als MAC PDU ins LAN. Nach Ablauf der üblichen Stationsverzögerungszeit kommt die MAC PDU sowohl bei der ACM von Station 1 als auch bei der ACM von Station 4 innerhalb der vorgegebenen „Zeitfenster“ ungestört an (siehe Bild 96g).

1.3.2.1.6 Der Zustand Await_ifm_response

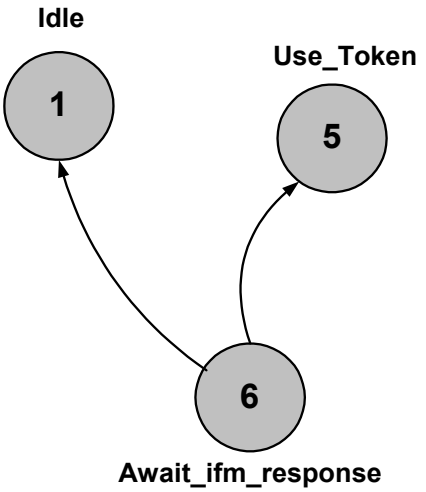
Befindet sich die ACM einer Station im Zustand Use_token und hat sie einer Empfänger-Station einen Daten-Frame geschickt, geht sie unmittelbar danach in den Zustand Await_ifm_response und wartet in diesem Zustand auf einen Response-Frame von der Empfänger-Station. Zur Erinnerung: Um das Realzeitverhalten des tokenbus-basierten LANs zu erhöhen, darf eine Empfänger-Station, nachdem sie im Zustand Idle einen Daten-Frame empfangen hat, einen Response-Frame an den Sender zurückschicken (siehe Bild 69a und 69b).
Welche Aktionen die Access Control Machine im Zustand Await_ifm_response ausführt, zeigen die folgenden Anweisungen:

Anweisungen:

```
case await_ifm_response:
   uart_in_empfangs_modus();

   response_window_timer=2*slot_time;
   empfange_acm_event(response_window_timer,&acm_event,&frame_control,
                              &dest_station,&source_station,&source_host);

   switch(acm_event)
   {
    case ok:
      if(frame_control == response && dest_station == this_station)
         acm_status = use_token;
      else
         acm_status = idle;
    break;
    default:                          /* timeout oder io_error */
         acm_status = use_token;
    break;
   }
break;
```

Sie schaltet als erstes den UART in den Empfangsmodus und initialisiert danach einen weiteren timer, nämlich den **response_window_timer** mit 2•slot_time.
Anschließend bringt die Access Control Machine mit **empfange_acm_event(...);** die ACMTask in den Wartezustand. Sie öffnet damit ein sogenanntes **response window**, also ein Zeitfenster, das die maximale Wartezeit auf den Response-Frame vorgibt. Die Überlegungen zur Bestimmung der Wartezeit im Zustand Await_ifm_response, sind die gleichen wie die im Zustand Check_pass_token, und führen auch hier zu 2•slot_time.

Transitionen



Welche Meldungen können bei der Access Control Machine im Zustand Await_ifm_response eintreffen? Betrachten wir als erstes die erfolgreiche Übertragung eines Response Frame gemäß Bild 97.

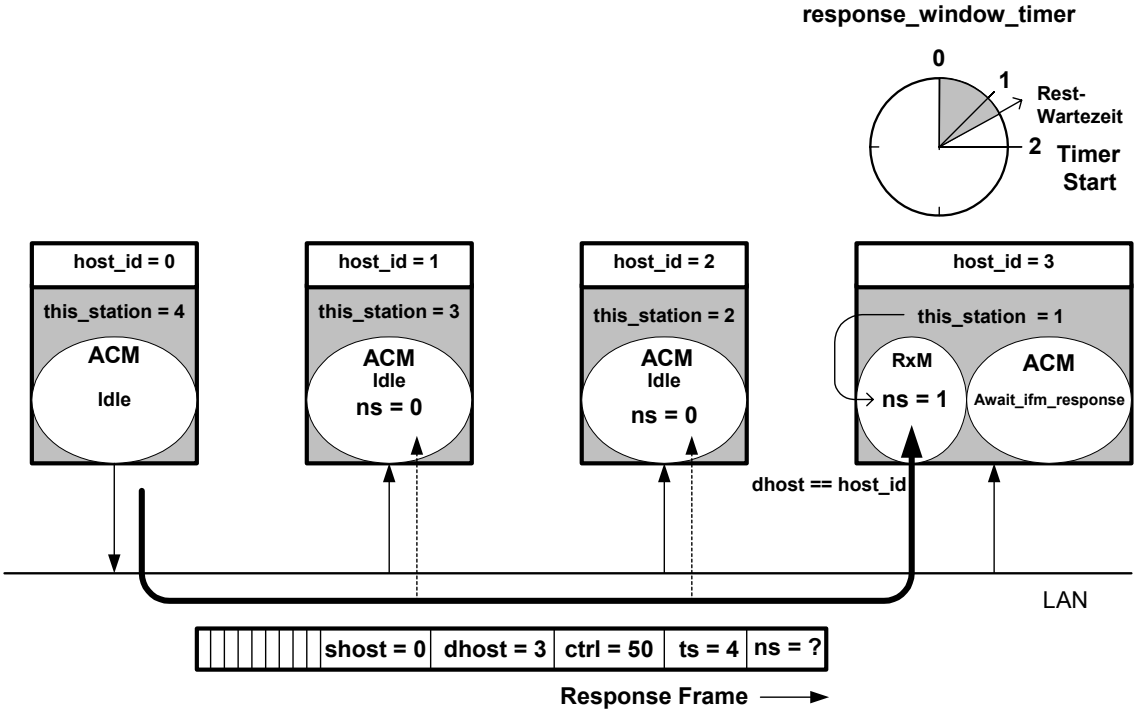


Bild 97: Empfang eines erfolgreich übertragenen Response Frame

Rechner 3 (host_id=3) hat im Zustand Use_token Rechner 0 (host_id=0) einen Daten-Frame geschickt. Er befindet sich jetzt im Zustand Await_ifm_response und wartet dort auf einen Response Frame von Rechner 0. Unmittelbar nach dem Übergang in den Zustand Await_ifm_response hat Rechner 3 den response_window_timer gestartet und bereits die übliche Stationsverzögerungszeit bis zur Ankunft des Daten-Frame bei Rechner 0 verbraucht. So steht jetzt noch ausreichend Rest-Wartezeit für die Ankunft des Response-Frame zur Verfügung. Wie Bild 97 zeigt, ist der Response Frame schon auf dem Weg.

Mit Ausnahme des Feldes **ns** (next_station), sind alle anderen Felder mit gültigen Informationen aufgefüllt. Wie behandelt Rechner 3 das ungültige ns-Feld ?
Wir erinnern uns: Jede Receive Machine (RxM) führt nach dem Empfang einer MAC PDU die Funktion **sende_acm_event(..)** ; aus (siehe Bild 50). Diese vergleicht das empfangene dhost-Feld mit der host_id des betreffenden Rechners und initialisiert bei Gleichheit das ns-Feld mit der Stationsnummer this_station. Bei Ungleichheit bekommt das ns-Feld den Wert 0.
In unserem Beispiel gemäß Bild 97 ist eine **ok**-Meldung bei der Receive Machine von Rechner 3 eingetroffen, und weil dort dhost == host_id ist, bekommt ns den Wert 1. Die Rechner 2 und 1 geben stattdessen ns folgerichtig den Wert 0.
Betrachten wir nur den Rechner 3. Seine Access Control Machine kann jetzt nach Ausführung der Funktion **empfange_acm_event(.....)** ; überprüfen, ob der Frame auch für sie «gedacht» ist. Denn nach dem Empfang der ok-Botschaft steht das ns-Feld zur Auswertung in der Variablen **dest_station** zur Verfügung.
Ist der Response Frame an sie adressiert (frame_control == response && dest_station == this_station), wechselt die ACM ihren Zustand nach **Use_token**. Dort kann sie nach der nächsten LLC PDU suchen, sie übertragen und wieder auf einen Response Frame warten usw. Dies geschieht solange, bis sie entweder keine LLC PDU mehr findet oder der token hold timer abgelaufen ist.

• Bekommt der Rechner 3 einen Daten-Frame (frame_control == data) oder einen Token-Frame (frame_control == token), dann „glaubt“ irgendeiner der verbleibenden Rechner im LAN, er sei im Besitz des Token. In diesem Fall geht die ACM über in den Zustand Idle und verhält sich dort passiv. Sie kann aber auch aktiv werden und vor dem Übergang in den Zustand Idle über **shost** den «Übeltäter» aufspüren und durch eine entsprechende Fehlermeldung die Beseitigung des Fehlers veranlassen.

• Eine andere (unwahrscheinliche) Situation wäre folgende: Rechner 3 schickt an Rechner 0 einen Daten-Frame. Rechner 0 befindet sich im Zustand Idle, identifiziert ihn und aktiviert daraufhin die Funktion **ifm_response(**unsigned char source_host, ..**);** um in der mit source_host korrespondierenden mailbox der Interface Machine-Sendeeinheit einen Response Frame festzustellen (siehe Bild 73). Sollte Rechner 0 aus irgendwelchen Gründen einen falschen source_host-Wert empfangen, dann sucht ifm_response() in der falschen mailbox nach einen potenziellen Response Frame und schickt ihn an den falschen **dhost** zurück.
Die Receive Machine von Rechner 3 stellt **dhost != host_id** fest und setzt **ns=0**. Damit empfängt die ACM zwar frame_control == response aber **dest_station != this_station**. Da dies nicht sein kann, geht sie in den Zustand Idle. Sie könnte aber, wie oben erwähnt, über shost die Fehlerquelle lokalisieren.
Kommt innerhalb der verbleibenden Wartezeit kein Response Frame zurück, weil z.B. der Rechner, der antworten sollte, nicht im Ring ist, dann behält die ACM das Token und geht über in den Zustand Use_token.
Kommt der Frame beschädigt an, interpretiert ihn die ACM als Response Frame (er hat lediglich den Weg durch das LAN nicht geschafft) und behält ebenfalls das Token.

1.3.2.1.7 Der Zustand Claim_token

Befindet sich eine Station im Zustand Idle und läuft der bus_idle_timer ab, ohne dass ein Token-Frame, ein Daten-Frame, ein Claim-Frame oder io_eror eintrifft, dann kann sich die Station nach Beendigung der Wartezeit um den Erhalt des Tokens bemühen. Sie verlässt deshalb den Zustand Idle und geht über in den Zustand **Claim_token**.
Dort begibt sie sich in einen Bewerbungs-Prozess, der nach dem Vorbild des Bit-Dominanz-Protokolls das Ziel verfolgt, andere Mitbewerber zur Aufgabe zu zwingen. Das Kernstück dieses Prozesses ist ein Algorithmus, der sich als **Interation** darstellt, d.h. einige Male wiederholt wird.
Mit jeder Ausführung erzeugt der Algorithmus einen Wartezyklus mit jeweils individueller Länge und schließt ihn mit einer Claim-Frame-Übertragung ab. Wie oft wird diese Prozedur wiederholt und wie wird sie bestimmt? Die Anzahl der Wiederholungen ist eine Funktion der **Stationsadresslänge** (gemessen in Bits) und ihre obere Grenze n berechnet sich zu
n = Adresslänge/2 + 1.
Angenommen, die Stationsadresse this_station ist eine 1-Oktet-Adresse (8 Bits), dann ist n = 8/2 + 1 = 5. Ist this_station eine 2-Oktet-Adresse (16 Bits), dann ist n = 16/2 + 1 = 9 usw.

Anmerkung: Wir werden später im Algorithmus die obere Grenze n durch die Variable **pass_count_max** ersetzen und den augenblicklichen Zählerstand durch die Variable **claim_pass_count** mit dem anfänglichen Wert 1 (untere Grenze) realisieren. So liegt die Anzahl der Wiederholungen im Bereich $1 \leq$ claim_pass_count $<$ pass_count_max.

Welche Längen haben die Wartezyklen und wie werden sie bestimmt? Die Wartezyklen leiten sich von der **Stationsadresse** ab und haben die Längen 0, 2, 4 oder 6 **slot_times**. Bevor wir klären, wie diese Längen zustande kommen, nehmen wir die Ergebnisse vorweg und sehen uns die individuellen Bewerbungs-Prozesse der Stationen R1, R2, R3 und R4 an. Siehe dazu die Bilder 98a und 98b.
Jede Station ist durch ihre 8 Bit-Adresse identifiziert, d.h. n=pass_count_max=5.

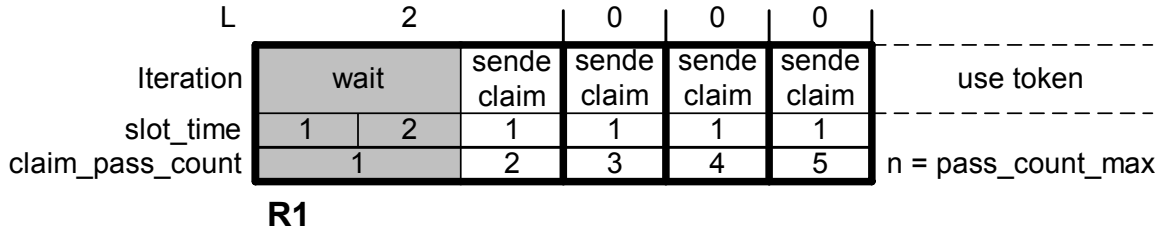


Bild 98a: Bewerbungs-Prozess der Station R1

Wie zu sehen ist, erzeugt die Iteration für R1 eine Kette von Wartezyklen mit den **L**(ängen) **2, 0, 0, 0** time_slots. Dabei werden die time_slots von einem weiteren Timer, nämlich dem **claim_timer** bereitgestellt.
Die Station R1 begibt sich im **1ten** Durchlauf (claim_pass_count=1) wegen L=2 für die Dauer von **2** slot_times in den Wartezustand. Während dieser Zeit „hört“ sie das LAN ab und wartet auf potenzielle Übertragungen von anderen Stationen.
Stellt sie eine Übertragung fest, das kann ein Claim-Frame, ein Data-Frame, ein Token-Frame oder ein io_error sein, gibt sie auf und geht in den Zustand Idle. In diesem Fall hat eine der anderen Stationen das Token gewonnen.

Stellt sie keine Übertragung fest, schickt sie einen Claim-Frame ins LAN und inkrementiert danach die Zählervariable (claim_pass_count=2). Sie hofft, dass alle anderen Stationen ihren Claim-Frame empfangen.
Ist dies der Fall, hat sie wahrscheinlich bereits zu diesem Zeitpunkt das Token gewonnen. Wenn nämlich die anderen Stationen den Claim-Frame von R1 empfangen, geben sie auf und gehen in den Idle-Zustand. Doch R1 kann das nicht wissen. Ihr Claim-Frame könnte ebenso gut mit den Claim-Frames anderer Stationen kollidiert sein. So kämpft sie weiter und begibt sich in den **2ten** Durchlauf.
Wegen L=0 ist die Dauer der Wartezeit **0** slot_times. Deswegen wartet sie nicht, sondern sendet unmittelbar den Claim-Frame ins LAN und inkrementiert danach die Zählervariable (claim_pass_count=3).
Sie begibt sich in den **3ten** Durchlauf, in dem sie wegen L=0 ebenfalls unverzüglich den Claim-Frame schickt und danach die Zählervariable inkrementiert (claim_pass_count=4). Und weil noch immer claim_pass_count < pass_count_max (4<5) ist, schickt sie in einem **4ten** Durchlauf einen weiteren Claim-Frame ins Netz. Wie üblich, inkrementiert sie danach die Zählervariable (claim_pass_count=5) und stellt fest, die Variable hat den Wert von pass_count_max erreicht. Damit ist der Bewerbungs-Prozess beendet. Die Iteration hat eine Kette von Wartezyklen mit den Längen L = 2, 0, 0, 0 erzeugt. Und weil die **Iteration zu Ende gebracht** werden konnte, gewinnt R1 das Token. Die Station verlässt daraufhin den Zustand Claim_token und geht in den Zustand Use_token.

Sehen wir uns als nächstes die Bewerbungs-Prozesse der restlichen Stationen R2, R3 und R4 an.

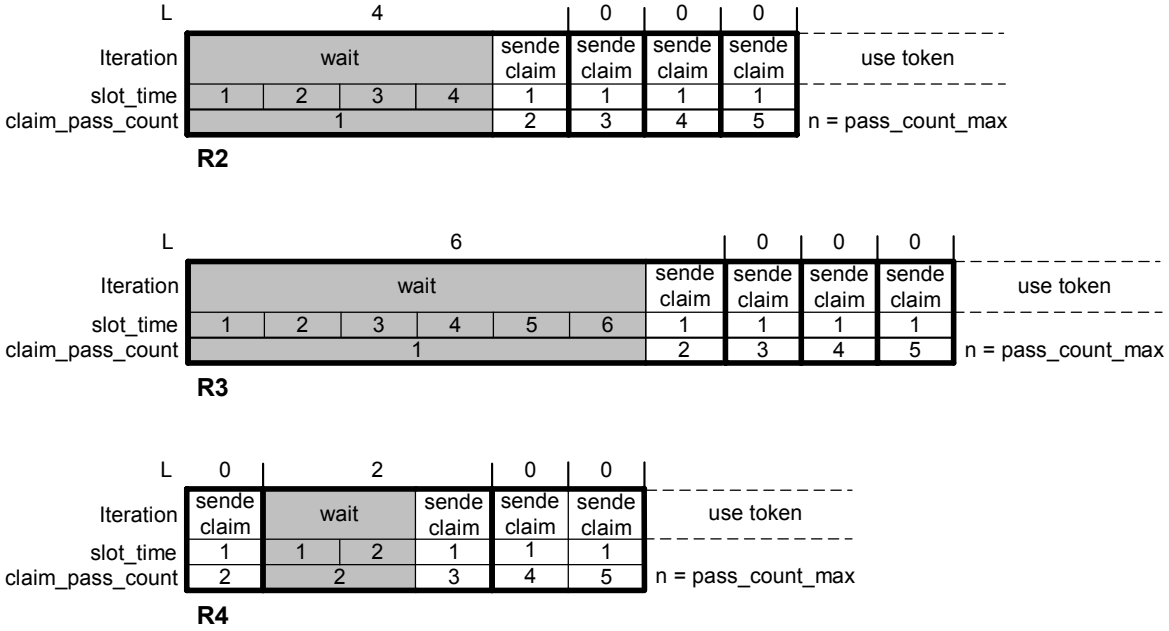


Bild 98b: Bewerbungs-Prozesse der Stationen R2, R3 und R4

Wie zu erkennen ist, erzeugt die Iteration die folgenden Ketten von Wartezyklen:

• R2 bekommt die Längen L = **4, 0, 0, 0** time_slots,
• R3 bekommt die Längen L = **6, 0, 0, 0** time_slots und
• R4 bekommt die Längen L = **0, 2, 0, 0** time_slots.

Sehen wir uns als nächstes an, wie der Claim_token-Algorithmus das Dominanz-Prinzip verwirklicht. Alle Stationen stehen in Konkurrenz zueinander und kämpfen um den Erhalt des Token. Zur Vereinfachung nehmen wir an, dass alle Stationen zur gleichen Zeit den Idle-Zustand verlassen und anschließend die Iteration beginnen. Siehe Bild 98c.

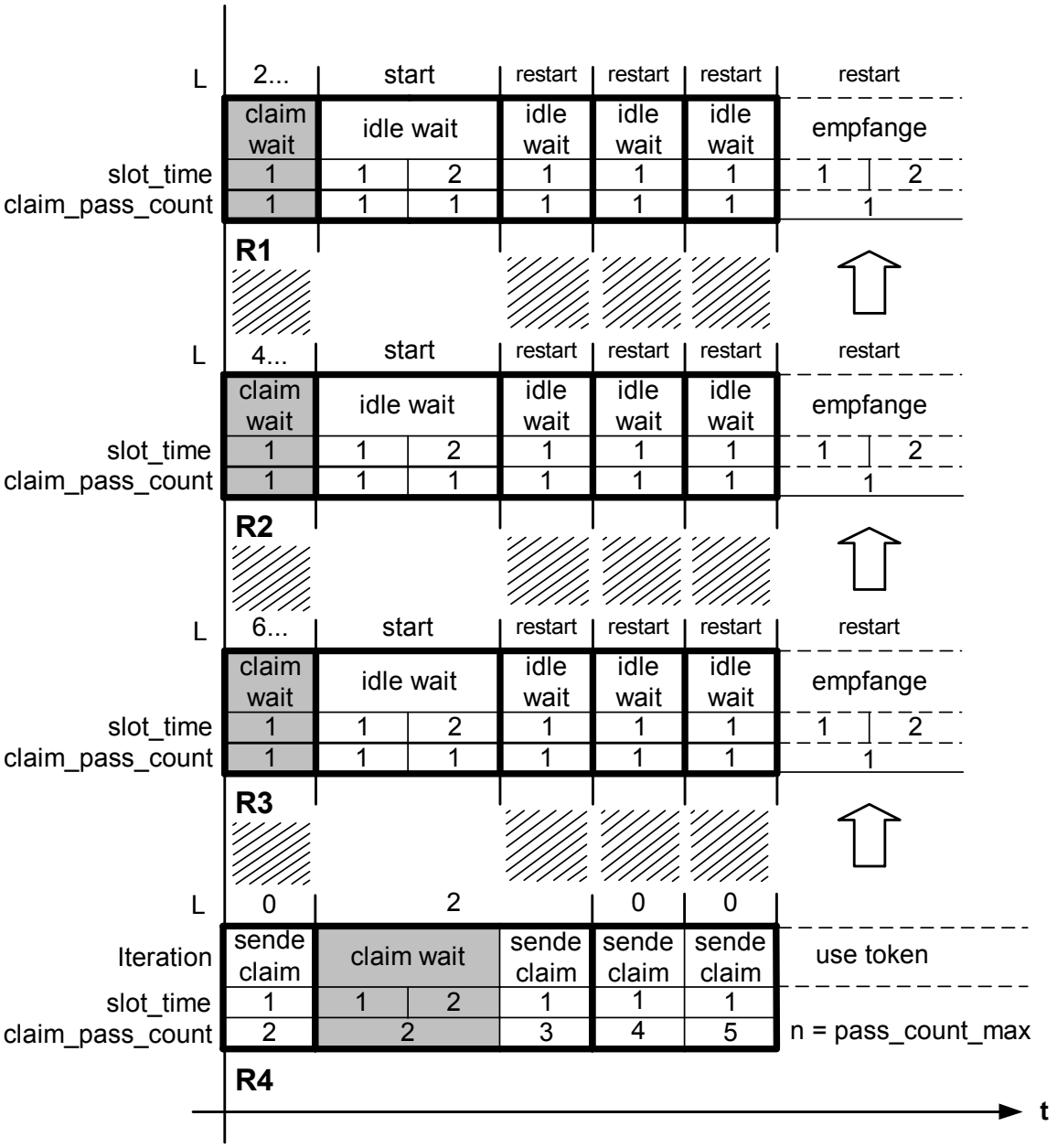


Bild 98c: Alle Stationen bewerben sich gleichzeitig um das Token

Die Stationen begeben sich anfänglich in den Wartezustand und führen „ihre“ sogenannten **claim waits** aus. Bei R1 sind es **2** slot times, bei R2 sind es **4** slot times, bei R3 sind es **6** slot times und bei R4 sind es **0** slot times. Bereits an dieser Stelle kann man einen guten Tipp abgeben, wer wohl das Token gewinnen wird. Weil R4 nicht wartet, sendet sie unverzüglich einen Claim-Frame, der innerhalb der slot time bei den anderen Stationen ankommt. R1, R2 und R3 gehen daraufhin in den Idle-Zustand und starten ihre bus_idle_timer.
Weil R4 nach „sende claim“ 2 claim waits ausführt, bleiben die restlichen Stationen 2 **idle waits** im Ruhezustand.

Nach Beendigung der beiden claim waits sendet R4 einen weiteren Claim-Frame ins Netz. R1, R2 und R3 empfangen ihn und starten daraufhin ihre bus_idle_timer von neuem (restart). Dies wiederholt sich, wenn R4 ein vorletztes Mal und ein letztes Mal seinen Claim-Frame überträgt. So bleiben R1, R2 und R3 im Idle-Zustand und werden auf diese Weise von R4 dominiert. R4 nimmt sich das Token, geht in den Zustand Use_token und schickt jetzt seine **Daten** ins Netz. Die Station mit der höchsten Adresse (this_station=4) hat das Token gewonnen.
Doch weil in der Realität die Stationen niemals gleichzeitig eingeschaltet werden können, verschieben sich die Ketten gegeneinander, und es zeigt sich, dass auch andere Stationen das Token gewinnen können.
Wir wollen nun klären, warum die maximale Wartezeit im Idle-Zustand 7 slot_times sein sollte: #define **idle_time 7*slot_time** und damit **bus_idle_timer=idle_time**.
Wir fassen die Station R3 ins Auge. Denn sie ist diejenige Station, bei der die Iteration mit **6 claim waits**, also mit der maximalen Anzahl beginnt. Während R3 im Zustand Claim_token seine claim waits ausführt, ist im ungünstigsten Fall eine andere Station (z.B. R2) im Zustand Idle parallel dabei im gleichen Zeitraster seine **idle waits** auszuführen. Siehe Bild 98d.

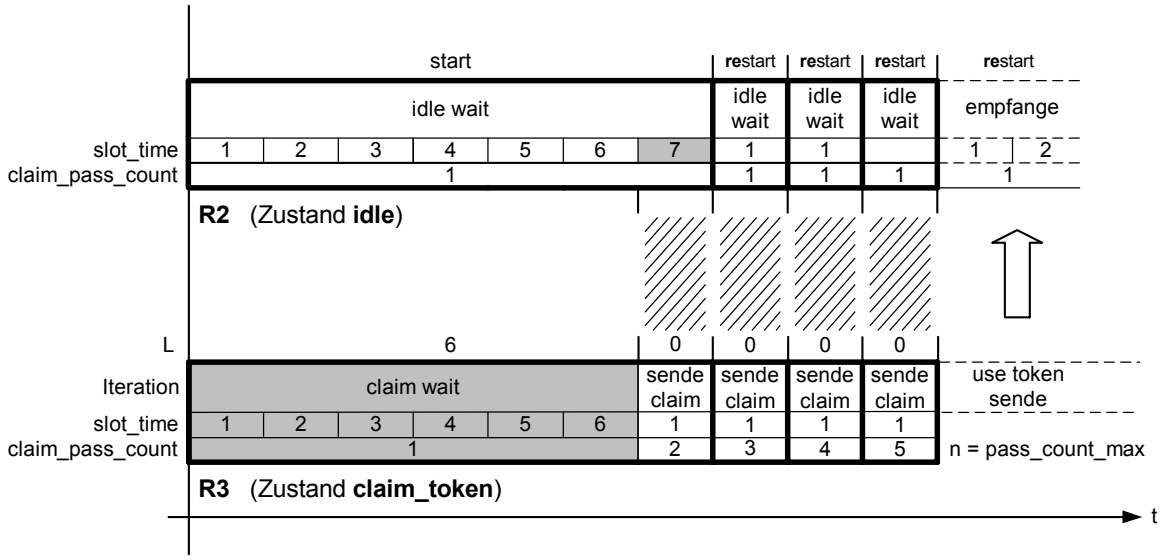


Bild 98d: Ungünstigster Fall bestimmt die Anzahl der idle waits

Sind 6 time_slots abgelaufen, überträgt R3 einen Claim-Frame ins LAN. Die Übertragung dauert 1 slot_time. Damit R2 den Claim-Frame identifizieren kann, sind somit **7 slot_times** erforderlich. So bleibt R2 im Idle-Zustand, und zwar solange, bis R3 auch seine restlichen Claim-Frames übertragen hat. Danach kann sich für R2 eine Zustandsänderung ergeben. Dies ist dann der Fall, wenn R3 Daten überträgt, die für R2 bestimmt sind.

Berechnung der claim wait-Längen L

Die claim wait-Längen L (0, 2, 4 oder 6) einer Station leiten sich aus den Bitwerten der betreffenden Stationsadresse ab. Um sie zu bekommen, müssen zunächst deren Bitpositionen bestimmt werden. Zu diesem Zweck betrachtet man jede **binäre** Stationsadresse als ein array, und unterteilt es in eine Anzahl von Elementen zu je **2** Adressbits.
Ist die Stationsadresse (this_station) eine 8 Bit-Adresse, dann ist die Anzahl der Elemente **a** = **Adresslänge/2** = 8/2 = **4**. Ist this_station eine 16 Bit-Adresse, dann ist a = 16/2 = **8**. Somit ist die Anzahl der Elemente identisch mit der maximalen Anzahl von Iterationen.

Es liegt deshalb auf der Hand, die Adress-Elemente des arrays über n=1 ... 4 (claim_pass_count=1 ... 4) zu indizieren. Dies ist im folgenden Bild 98e am Beispiel von 8 Bit-Stationsadressen illustriert.

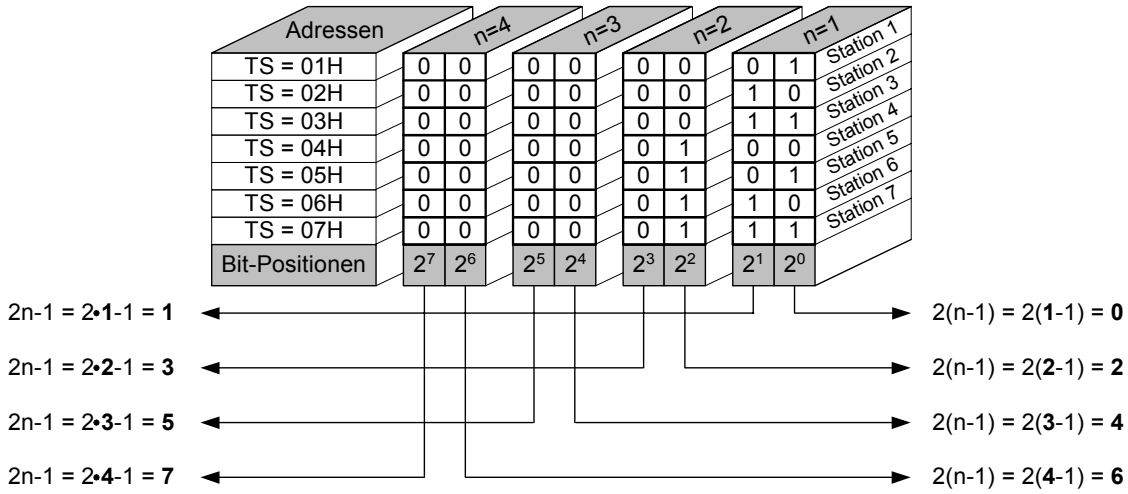


Bild 98e: Ermittlung der Bitpositionen

Wie zu sehen ist, sind die Adressen von 7 Stationen in hexadezimaler und binärer Form dargestellt. Sie lauten TS (this_station) = 01H ... 07H bzw. $0000\ 0001_2$ ... $0000\ 0111_2$.
Ist n=1, dann ist dasjenige Adress-Element erreichbar, das die Adressbits an den Positionen 2^1 und 2^0 enthält.
Ist n=2, erreicht man dasjenige Adress-Element, das die Adressbits an den Positionen 2^3 und 2^2 enthält usw.

Innerhalb eines Adress-Elements hält sich
- das LSB (Least Significant Bit) an einer geraden Position auf (ausgedrückt durch die Exponenten 0, 2, 4 und 6) und
- das MSB (Most Significant Bit) an einer ungeraden Position (ausgedrückt durch die Exponenten 1, 3, 5 und 7).

Eine **gerade** Position lässt sich nun als Funktion von n wie folgt berechnen: **2(n-1)**.
Beispiel: Ist n=1 ergibt sich die Adressbit-Position 2(1-1) = **0**.
Ist n=2 ergibt sich die Adressbit-Position 2(2-1) = **2** usw. (vgl. mit Bild 98e).

Eine **ungerade** Position lässt sich als Funktion von n wie folgt berechnen: **2n-1**.
Beispiel: Ist n=1 ergibt sich die Adressbit-Position 2•1-1 = **1**.
Ist n=2 ergibt sich die Adressbit-Position 2•2-1 = **3** usw. (vgl. mit Bild 98e).

In allen Adress-Elementen (n=1 ... 4) einer jeden Station bekommen wir die **Bitwerte** an den **geraden** Positionen mit **Bit(2(n-1))**.
Beispiel Station 1:
Ist n=1 bekommen wir an der Position 2(1-1)=0 den Bitwert **1**.
Ist n=2 bekommen wir an der Position 2(2-1)=2 den Bitwert **0** usw.
(vgl. mit Bild 98e).

Ähnliches gilt für die Bitwerte an den **ungeraden** Positionen. Wir bekommen sie mit **2•Bit(2n-1)**.

Beispiel Station 1:

Ist n=1 bekommen wir an der Position 2•1-1=1 den Bitwert 2•0 = **0**.
Ist n=2 bekommen wir an der Position 2•2-1=3 den Bitwert 2•0 = **0** usw.
(vgl. mit Bild 98e).

In der folgenden Grafik sind für die **Station 1** alle Bitwerte für n=1 .. 4 zusammengestellt. Desweiteren sind die daraus resultierenden claim wait-Längen L eingetragen. Sie berechnen sich zu **L = 2•Bitwert**. Ihre Werte L=2, 0, 0, 0 haben wir bereits benutzt. Wir finden sie wieder im Bewerbungs-Prozess der Station R1 nach Bild 98a.

Station 1

n	(2n-1)	2(n-1)	2• Bit(2n-1) + Bit(2(n-1))	Bitwert	L= 2•Bitwert
1	1	0	2•0 = 0 + 1	1	2
2	3	2	2•0 = 0 + 0	0	0
3	5	4	2•0 = 0 + 0	0	0
4	7	6	2•0 = 0 + 0	0	0

Bild 98f: Ermittlung der Bitwerte und Berechnung der claim wait-Längen für Station 1

Die Berechnung der L-Werte für die restlichen Stationen 2 ... 7 sind in den folgenden Grafiken zu sehen. Wobei wir die Werte der Stationen 2, 3 und 4 ebenfalls schon benutzt haben. Sie sind in den Bewerbungs-Prozessen der Stationen R2, R3 und R4 nach Bild 98b wieder zu finden. Die der Station 2 lauten L=4, 0, 0, 0, die der Station 3 L= 6, 0, 0, 0 und die der Station 4 L=0, 2, 0, 0.

Station 2

n	(2n-1)	2(n-1)	2• Bit(2n-1) + Bit(2(n-1))	Bitwert	L= 2•Bitwert
1	1	0	2•1 = 2 + 0	2	4
2	3	2	2•0 = 0 + 0	0	0
3	5	4	2•0 = 0 + 0	0	0
4	7	6	2•0 = 0 + 0	0	0

Station 3

n	(2n-1)	2(n-1)	2• Bit(2n-1) + Bit(2(n-1))	Bitwert	L= 2•Bitwert
1	1	0	2•1 = 2 + 1	3	6
2	3	2	2•0 = 0 + 0	0	0
3	5	4	2•0 = 0 + 0	0	0
4	7	6	2•0 = 0 + 0	0	0

Station 4

n	(2n-1)	2(n-1)	2•Bit(2n-1) + Bit(2(n-1))	Bitwert	L= 2•Bitwert
1	1	0	2•0 = 0 + 0	0	0
2	3	2	2•0 = 0 + 1	1	2
3	5	4	2•0 = 0 + 0	0	0
4	7	6	2•0 = 0 + 0	0	0

Station 5

n	(2n-1)	2(n-1)	2•Bit(2n-1) + Bit(2(n-1))	Bitwert	L= 2•Bitwert
1	1	0	2•0 = 0 + 1	1	2
2	3	2	2•0 = 0 + 1	1	2
3	5	4	2•0 = 0 + 0	0	0
4	7	6	2•0 = 0 + 0	0	0

Station 6

n	(2n-1)	2(n-1)	2•Bit(2n-1) + Bit(2(n-1))	Bitwert	L= 2•Bitwert
1	1	0	2•1 = 2 + 0	2	4
2	3	2	2•0 = 0 + 1	1	2
3	5	4	2•0 = 0 + 0	0	0
4	7	6	2•0 = 0 + 0	0	0

Station 7

n	(2n-1)	2(n-1)	2•Bit(2n-1) + Bit(2(n-1))	Bitwert	L= 2•Bitwert
1	1	0	2•1 = 2 + 1	3	6
2	3	2	2•0 = 0 + 1	1	2
3	5	4	2•0 = 0 + 0	0	0
4	7	6	2•0 = 0 + 0	0	0

Bild 98g: Ermittlung der Bitwerte und Berechnung der claim wait-Längen für die Stationen 2 ... 7

Der claim token-Algorithmus

Damit der claim token-Algorithmus die spezifischen **claim waits** der Stationen bilden kann, braucht er die oben ermittelten L-Werte. Um ihn von diesen Berechnungen zu entlasten, speichern wir sie in einer Matrix mit dem Namen L. Ihr Layout ist im folgenden Bild zu sehen:

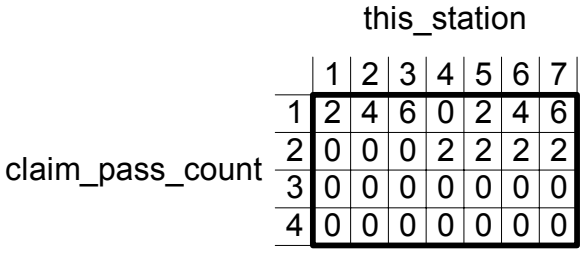


Bild 98h: Matrix enthält die L-Werte der Stationen

Jeder Station (this_station) ist eine Spalte zugeordnet, in der die L-Werte der betreffenden Station aufbewahrt sind. Vgl. mit den Bildern 98f und 98g. So kann der claim token-Algorithmus mit der Variablen this_station die mit „seiner" Station assoziierte Spalte indizieren. Und während er sie „festhält", kann der Zeilenindizierer claim_pass_count der Reihe nach die Eintragungen in der betreffenden Spalte erreichen.
Die Initialisiserung der Matrix übernimmt entweder die Betriebssytem-Funktion init() oder die MAC-Funktion acm_exe() in folgender Weise:

```
#define   pass_count_max      5
#define   station_max         5

                •
                •

static unsigned char      L[pass_count_max][station_max] = {0,0,0,0,0,
                                                            0,2,4,6,0,
                                                            0,0,0,0,2,
                                                            0,0,0,0,0,
                                                            0,0,0,0,0};
```

Bild 98i: Initialisierung der Matrix L

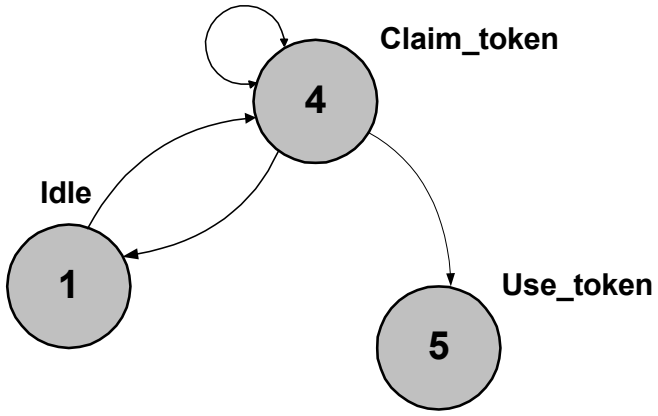
Anmerkung: Weil in einem tokenbus-basierten LAN die Stationsnummer 0 **nicht** vorkommt, ist die Spalte 0 mit Nullen aufgefüllt. Ähnliches gilt für die Zeile 0. Weil der Wiederholungszähler claim_pass_count erst ab 1 zu zählen beginnt, ist die Zeile 0 ebenfalls mit Nullen aufgefüllt.

Alle Vorarbeiten sind nun abgeschlossen, so dass wir uns jetzt dem Algorithmus zuwenden können. Wie er sich gestaltet, ist zusammen mit den resultierenden Transitionen in den folgenden Bildern zu sehen:

Anweisungen

```
case claim_token:
 uart_in_empfangs_modus();

 if(claim_pass_count < pass_count_max)
   {
    claim_timer = L[claim_pass_count][this_station]*slot_time;
    empfange_acm_event(claim_timer,&acm_event,&frame_control,
                              &dest_station,&source_station,&source_host);
    switch(acm_event)
    {
     case timeout:
       uart_in_sende_modus();
       mac_pdu[ctrl] = claim;
       txm_exe(mac_pdu);
       claim_pass_count++;
     break;

     default: acm_status = idle;       /* claim, data, token, io_error */
     break;
    }
  }
  else                                  /*    claim_pass_count == 5   */
     {
      set_thtr(hi_pri_token_hold_time);
      acm_status            = use_token;
     }
break;
```

Transitionen



Die Access Control Machine kann den Zustand Claim_token nur durch einen Übergang vom Zustand Idle aus erreichen. Wenn in diesem Zustand der claim token-Algorithmus die Iteration ausführt, bleibt die ACM während der Wiederholungen in diesem Zustand. Kann der Algorithmus die Iteration zu Ende bringen, hat die Station das Token gewonnen und die ACM geht über in den Zustand Use_token. Gelingt das nicht (weil eine andere Station „stärker“ ist), geht die ACM zurück in den Zustand Idle.

Der claim token-Algorithmus schaltet als erstes den UART in den Empfangs-Modus. Danach stellt er mit if(claim_pass_count < pass_count_max) fest, ob die Iteration noch im Gang ist.

- Wenn **nein**, hat claim_pass_count den Maximalwert pass_count_max = 5 erreicht. Er verlässt deshalb den if-Block und geht in den else-Block.
  Dort setzt er mit set_thtr(hi_pri_token_hold_time); den **token hold timer** und bestimmt mit acm_status = use_token; den neuen ACM-Zustand.
- Wenn **ja**, ist die Iteration noch im Gang. Deshalb holt sich der Algorithmus mit L[claim_pass_count][this_station] den aktuellen **L**(ängen)-Wert aus der L-Matrix und bildet daraus mit claim_timer = L[claim_pass_count][this_station]*slot_time; die Anzahl der Wartezyklen. Danach geht die ACM mit empfange_acm_event(**claim_timer** ...); in den Wartezustand. Dort kann sie über **acm_event** vier verschiedene Botschaften empfangen, die der claim token-Algorithmus mit switch(acm_event) in zwei Kategorien unterteilt: **timeout** und **default**. Letztere bedeutet, dass während der Wartezeit entweder ein Claim-Frame, ein Data-Frame, ein Token-Frame oder ein io_error eingetroffen ist. In diesem Fall muss die ACM zurück in den Zustand Idle und der Algorithmus bestimmt mit acm_status = idle; den neuen Zustand.
  Kommt am Ende der Wartezeit die Nachricht **timeout**, dann ist offensichtlich im aktuellen Iterationsschritt **keine** „stärkere“ Station im Netz. Bei einem Wartezyklus von 0(ms) folgert dies der Algorithmus sogar unverzüglich. Denn bei claim_timer=0 beendet der Zeitmechansismus der ACM sofort den Wartezustand.
  Ob nun verzögert oder unverzüglich, am Ende der Wartezeit schaltet der Algorithmus den UART in den Sende-Modus, schickt mit mac_pdu[ctrl] = claim; und txm_exe(mac_pdu); einen Claim-Frame ins Netz und inkrementiert danach mit claim_pass_count++; den Wiederholungszähler. Bleibt die ACM im Zustand Claim_token, geschieht dies solange, bis der Zähler den Maximalwert pass_count_max erreicht hat. Dann bricht die anfängliche if-Abfrage die Iteration ab.

Nachdem nun die Aktionen und Algorithmen der ACM-Zustände im Einzelnen dargestellt worden sind, zeigt zur Abrundung dieses Abschnitts das folgende Bild das resultierende komplette Programm der Betriebssystem-Funktion **acm_exe()**.

```
#define idle                        1
#define claim_token                 4
#define use_token                   5
#define await_ifm_response          6
#define pass_token                  8
#define check_pass_token            9
#define mac_pdu_max                14
#define claim                       0
#define token                    0x08
#define data                     0x40
#define response                 0x50
#define timeout                    -2
#define io_error                   -1
#define pass_count_max              5
#define station_max                 5
#define slot_time                  50
#define idle_time          7*slot_time
#define ok                          0
#define ns                          0
#define ts                          1
#define ctrl                        2
#define max_access_class            6
#define enable                      1
#define disable                     0

extern unsigned char          mac_pdu[mac_pdu_max];
extern unsigned char          send_pending[max_access_class+1];
extern unsigned char          llc_pdu_pending;
extern unsigned char          first_station;
extern unsigned char          this_station;
extern unsigned char          next_station;
extern unsigned char          next_init;
extern char                   acm_status;
extern char                   frame_control;
extern unsigned short int     token_hold_timer;
extern unsigned short int     hi_pri_token_hold_time;

static char                   acm_event;
static unsigned char          dest_station;
static unsigned char          source_station;
static unsigned char          next_station;
static unsigned char          source_host;
static unsigned char          claim_pass_count;
static unsigned short int     bus_idle_timer;
static unsigned short int     claim_timer;
static unsigned short int     token_pass_timer;
static unsigned short int     response_window_timer;
static unsigned char          L[pass_count_max][station_max]={0,0,0,0,0,
                                                              0,2,4,6,0,
                                                              0,0,0,0,2,
                                                              0,0,0,0,0,
                                                              0,0,0,0,0};
```

Bild 98j: Betriebssystem-Funktion acm_exe(); (Datei acmexe.c)

```
void acm_exe()
{
 switch(acm_status)
 {
  case idle:
    clear_thtr();
    claim_pass_count = 1;
    uart_in_empfangs_modus();

    bus_idle_timer=idle_time;
    empfange_acm_event(bus_idle_timer,&acm_event,&frame_control,
                       &dest_station,&source_station,&source_host);
    switch(acm_event)
    {
     case ok:
      switch(frame_control)
      {
       case token:
        if(dest_station==this_station)
          {                                          /*   receive_token   */
           set_thtr(hi_pri_token_hold_time);
           acm_status = use_token;
          }
       break;
       case data:                                   /* transmit_response */
          if(dest_station == this_station)
            {
             ifm_response(source_host,&llc_pdu_pending,mac_pdu);
             if(llc_pdu_pending)
               {
                mac_pdu[ns]   = source_station;
                mac_pdu[ts]   = this_station;
                mac_pdu[ctrl] = response;
                uart_in_sende_modus();
                txm_exe(mac_pdu);
               }
            }
       break;
       default: break;
      }
     break;

     case timeout:
       acm_status = claim_token;
     break;

     default: break;                                /* io_error */
    }
  break;
```

Bild 98j: Betriebssystem-Funktion acm_exe(): Fortsetzung

```
case claim_token:
 uart_in_empfangs_modus();

 if(claim_pass_count < pass_count_max)
   {
   claim_timer = L[claim_pass_count][this_station]*slot_time;
   empfange_acm_event(claim_timer,&acm_event,&frame_control,
                &dest_station,&source_station,&source_host);
   switch(acm_event)
   {
    case timeout:
     uart_in_sende_modus();
     mac_pdu[ctrl]=claim;
     txm_exe(mac_pdu);
     claim_pass_count++;
    break;

    default: acm_status = idle;       /* claim, data, token, io_error */
    break;
   }
 }
 else                                  /*    claim_pass_count == 5   */
    {
     set_thtr(hi_pri_token_hold_time);
     acm_status            = use_token;
    }
break;

case use_token:
 uart_in_sende_modus();
 ifm_round_robin_read(&send_pending[max_access_class],mac_pdu);
 if(send_pending[max_access_class] && token_hold_timer > 0)
   {
   get_tht(&token_hold_timer);                    /*    send_frame     */

   mac_pdu[ts]   =  this_station;
   mac_pdu[ctrl] =  data;
   txm_exe(mac_pdu);

   acm_status     = await_ifm_response;
   }
 else
   acm_status    = pass_token;
break;
```

Bild 98j: Betriebssystem-Funktion acm_exe(): Fortsetzung

```
case await_ifm_response:
   uart_in_empfangs_modus();

   response_window_timer=2*slot_time;
   empfange_acm_event(response_window_timer,&acm_event,&frame_control,
                              &dest_station,&source_station,&source_host);

   switch(acm_event)
   {
    case ok:
     if(frame_control == response && dest_station == this_station)
        acm_status  =  use_token;
     else
        acm_status  =  idle;
     break;
     default:                          /* timeout oder io_error */
        acm_status  =  use_token;
     break;
   }
break;

case pass_token:
 uart_in_sende_modus();
 mac_pdu[ns]    =  next_station;              /*   pass_token    */
 mac_pdu[ts]    =  this_station;
 mac_pdu[ctrl]  =  token;
 txm_exe(mac_pdu);

 acm_status     =   check_pass_token;
break;
```

Bild 98j: Betriebssystem-Funktion acm_exe(): Fortsetzung

```
case check_pass_token:
  uart_in_empfangs_modus();

  token_pass_timer  = 2*slot_time;
  empfange_acm_event(token_pass_timer,&acm_event,&frame_control,
                          &dest_station,&source_station,&source_host);

  switch(acm_event)
  {
   case timeout:                          /* pass_token_failed */
       acm_status = pass_token;
       next_station--;

       if(next_station == 0)
         next_station =  first_station;

       if(next_station == this_station)
         {
         next_station =  next_init;
         acm_status   =  idle;
         }
    break;

    default:                              /*  ok oder io_error  */
       acm_status   = idle;               /*        pass ok        */
       next_station = next_init;
    break;
   }
  break;
}
}
```

Bild 98j: Betriebssystem-Funktion acm_exe(): Fortsetzung

1.3.3 Die Transmit Machine (TxM)

Diese logische Einheit des MAC Layers akzeptiert einen Frame von der Access Control Machine, formt daraus eine MAC Protokol-Dateneinheit (MAC PDU) und überträgt diese als eine Sequenz von Oktets an den UART. Sie kann als Task oder als Funktion realisiert sein, und sie liegt innerhalb der MAC-Struktur als Interface zwischen der ACM und dem UART. In der vorliegenden Realisierung ist die TxM als Funktion implementiert. Ihr Name lautet **TxM_exe(unsigned char acm_pdu[])** und sie wird mit der Adresse des array mac_pdu[] von der ACM in den Zuständen **Idle**, **Use_token**, **Claim_token** oder **Pass_token** aufgerufen. Siehe Bild 99.

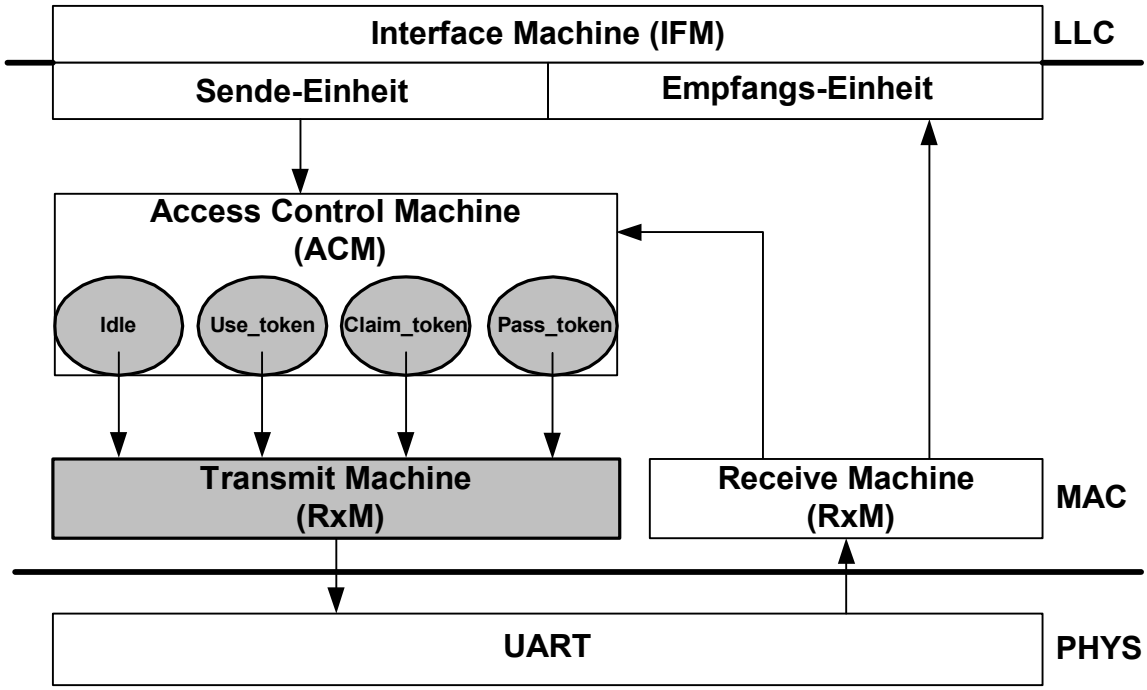


Bild 99 : Aufruf der Transmit Machine

Welche Aktionen die Funktion im einzelnen ausführt, zeigt der folgende Programmcode im Bild 101. Als erstes berechnet die Funktion die fcs des zu übertragenden Frames. Abhängig vom Zustand der ACM bekommt die Funktion vier verschiedene Typen von Frames. Siehe Bilder 100a..100d.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
ns=?	ts	ctr=40	dhost	shost	seq	ack	dport	sport	len	daten		fcs = ?	

Bild 100a: Daten-Frame (ctr=40) aus Zustand Use_token

0	1	2	3	4	5	6	7	8	9	10	11	12	13
ns	ts	ctr=50	dhost	shost	seq	ack	dport	sport	len	daten		fcs = ?	

Bild 100b: Response-Frame (ctr=50) aus Zustand Idle

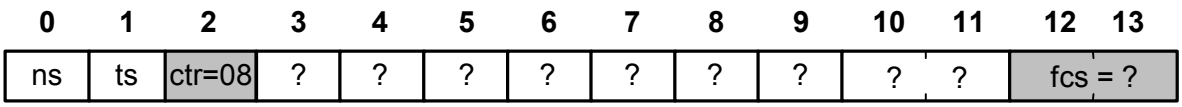


Bild 100c: Token-Frame (ctr=08) aus Zustand Pass_token

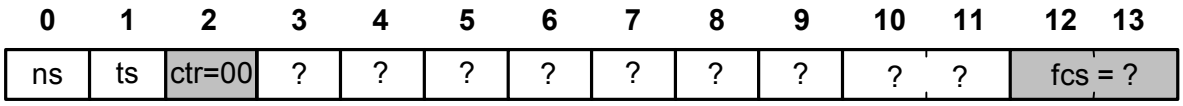


Bild 100d: Claim-Frame (ctr=00) aus Zustand Claim_token

```
#inlude<i86.h>

#define acm_pdu_max          14
#define mask               0x1021
#define thr1                0x380
#define lsr1                0x385
#define fcr1                0x382

extern unsigned short int crc(unsigned short int,unsigned short int,
                              unsigned short int);

static unsigned short int   fcs;
static unsigned char          i;

void TxM_exe(unsigned char acm_pdu[])
{
 fcs=0;                                          /*     fcs berechnen       */
 for(i=0;i<(acm_pdu_max-2);i++)
     fcs=crc(fcs,acm_pdu[i],mask);

 acm_pdu[12] = (unsigned char) (fcs>>8) & 0xff;   /*     fcs-high Byte       */
 acm_pdu[13] = (unsigned char) fcs & 0xff;        /*     fcs-low Byte        */

 for(i=0;i<acm_pdu_max;i++)
      outbyte(thr1,acm_pdu[i]);                   /*   Frame an UART         */
 while((inbyte(lsr1) & 0x40) == 0);               /*   alle bytes gesendet   */

 outbyte(fcr1,0xcd);                             /* Transmitter-FIFO Reset */
}
```

Bild 101: Betriebssystem-Funktion TxM_exe (Datei txmexe.c)

Jeder Frame hält sich im array acm_pdu[acm_pdu_max] auf, das aus 14 Bytes besteht. Die letzten 2 Bytes (12 und 13) sind für die fcs reserviert. Und so berechnet die Funktion TxM_exe(.); die fcs für die ersten 12 Bytes (acm_pdu_max-2) und trägt anschließend das 16 Bit-Ergebnis (Typ unsigned short int) in das reservierte fcs-Feld ein. In welcher Byte-Reihenfolge muß der Eintrag erfolgen? Antwort: Das High-Byte der fcs in Byte 12 des arrays und das Low-Byte in Byte 13 (vergleiche mit Bild 29: Bitstrom für n=14 Oktets).

- Mit **acm_pdu[12] = (unsigned char) (fcs>>8) & 0xff;** wird bei Nutzung des casting-Mechanismus das High-Byte der fcs extrahiert und gespeichert und
- mit **acm_pdu[13] = (unsigned char) fcs & 0xff**; das Low-Byte.

Nachdem dies geschehen ist, legt die Transmit Machine in einer for-schleife die Elemente des arrays über das sogenannte **t**ransmitter **h**olding **r**egister (**thr**) Byte für Byte im FIFO des UART ab.

Der Zugriff auf den FIFO erfolgt mit outbyte(thr1,acm_pdu[i]); wobei thr1 die Adresse des ihm vorgeschalteten transmitter holding registers ist:

```
for(i=0;i<acm_pdu_max;i++)
    outbyte(thr1,acm_pdu[i]);   /*     Frame an UART       */
```

Wir wissen, der FIFO ist als Ringpuffer organisiert. Beginnend ab Byte 0 indiziert er nach jedem Schreibzyklus das nächste Byte im FIFO automatisch und schickt den Inhalt des FIFO Byte für Byte ins LAN.
Die Funktion überwacht, ob auch alle Bytes ausgesendet worden sind. Zu diesem Zweck begibt sie sich in die nachfolgende while-Schleife und verfolgt dort in einem sogenannten „busy waiting“ die Vorgänge im FIFO:

```
while((inbyte(lsr1) & 0x40) == 0);   /*     alle bytes gesendet?    */
```

In dieser Schleife liest sie mit inbyte(lsr1) das line status register, maskiert mit & 0x40 das Bit 6 (transmitter empty) und bekommt über den resultierenden Wert dieses Bits Auskunft über den Zustand des Transmitters.

- Ist Bit 6 = 0, ist der FIFO **nicht leer** und die Funktion bleibt in der Schleife. Die Bytes sind noch nicht alle ausgesendet.
- Ist Bit 6 = 1, ist der FIFO leer und die Funktion verläßt die Schleife. Alle Bytes sind ausgesendet.

Hat die Funktion die while-Schleife verlassen, bringt die Transmit Machine den FIFO mit

```
outbyte(fcr1,0xcd);
```

wieder in einen definierten Anfangszustand; fcr1 ist die Adresse des FIFO-Control-Registers und 0xcd ist der Reset Code des Transmitter FIFO.