

1. Begriffe

Bit Das steht eigentlich für "binary digit" und bezeichnet in der Informatik eine Maßeinheit für Datenmengen. Dabei ist 1Bit die kleinste Einheit und kann zwei Zustände enthalten - 0 oder 1.

Signal Zur Kommunikation über ein Medium muss eine Bitsequenz in ein physikalisch übertragbares Muster umgewandelt werden, das sogenannte Signal. Dieses Signal ist allerdings äußeren Einflüssen unterworfen, so dass es korumpiert werden kann.

Multiplexing Als Multiplexing wird eine Methode der Signalübertragung bezeichnet, bei der mehrere Signale gebündelt werden und gleichzeitig über ein Medium verschickt werden. So erlaubt Multiplexing ein effizienteres Nutzen des Mediums.

Framing Das ist das Verfahren, wie ein physischer Bitstrom in eine Sequenz von Frames, also wohlabgegrenzte Einheiten.

Header Um beim Framing die einzelnen Frames unterscheiden zu können und zu wissen, wo ein Frame aufhört bzw. anfängt, packt man diese Informationen über den Frame einfach an den Anfang jedes Frames, also Header. Der Header kann ganz unterschiedlich aussehen, z.B. kann er einfach nur die Anzahl der Bits im nächsten Frame angeben oder ziemlich komplex sein.

Piggybacking Beim Piggybacking wird die Bestätigung des Erhalt einer Information vom Sender mit in die Headerinformation des Frames gesteckt, der zurückgesendet wird.

2. Kodierungsverfahren

a) Aus dem Generatorpolynom $x^2 + 1$ ergibt sich die Bitsequenz 101. Jetzt muss also zu der zu übertragenden Folge noch 00 angehängt werden und dann per Polynomdivision die Frame Check Sequenz berechnet.

```
111100100 : 101
101
---
0101
101
---
00000100
101
---
001
```

Da die Länge des FCS dem Grad des Generatorpolynoms entspricht, wird also noch 01 an die Bitfolge rangehängen. Es muss also 111100101 mit Manchester codiert werden. Hierbei steht die steigende Flanke, also 01 für eine 1 und die 10 für eine 0. Es ergibt sich demnach

010101011010011001. Da der Frame per 111 markiert wird, muss geguckt werden, dass auf keinen Fall mehr als zwei Einsen hintereinander stehen, sonst wäre der Frame nicht mehr eindeutig. Es ist also Stuffing nötig um dies zu gewährleisten, da einmal eine 11 auftritt. Mit Framing und Bitstuffing sieht die Folge dann letztendlich wie folgt aus:

111 0101010110100110001 111

b) Das Generatorpolynom ist wieder $x^2 + 1$, das zugehörige Bitmuster also wieder 101. Um zu gucken, ob ein Fehler bei der Übertragung aufgetreten ist, muss die empfangene Bitfolge per Polynomdivision durch das Generatorpolynom geteilt werden. Dabei gibt dann der auftretende Rest an, ob ein Fehler auftrat oder nicht. Wenn nämlich der Rest 0 ist, war alles super, ansonsten wissen wir, dass es einen Fehler gab.

```
1001110001010110 : 101
101
---
00111
101
---
0101
101
---
000000101
101
---
0000110
101
---
011
```

Der Rest ist also ungleich null, somit ist also ein Fehler aufgetreten.

3. Datenrate

Bei einem rauschlosen Kanal kann die Formel von Nyquist verwendet werden um die Datenrate zu berechnen:

$$datRate \leq 2 * 5 * 10^6 Hz * \log_2 6 \frac{bits}{s} = 25,85 * 10^6 \frac{bits}{s}$$

4. Fifty/Fifty

- UDP Pakete werden aufgrund des nicht benötigten Verbindungsaufbaus bevorzugt für Multimedia- Anwendungen verwendet.

Falsch. UDP wird bevorzugt verwendet, da es nicht auf die Korrektheit der Nachricht achtet und hierbei durch nicht wiederholtes Senden etc. Zeit einspart. Allerdings ist es auch ein verbindungsloses Protokoll und muss keine Verbindung extra aufbauen.

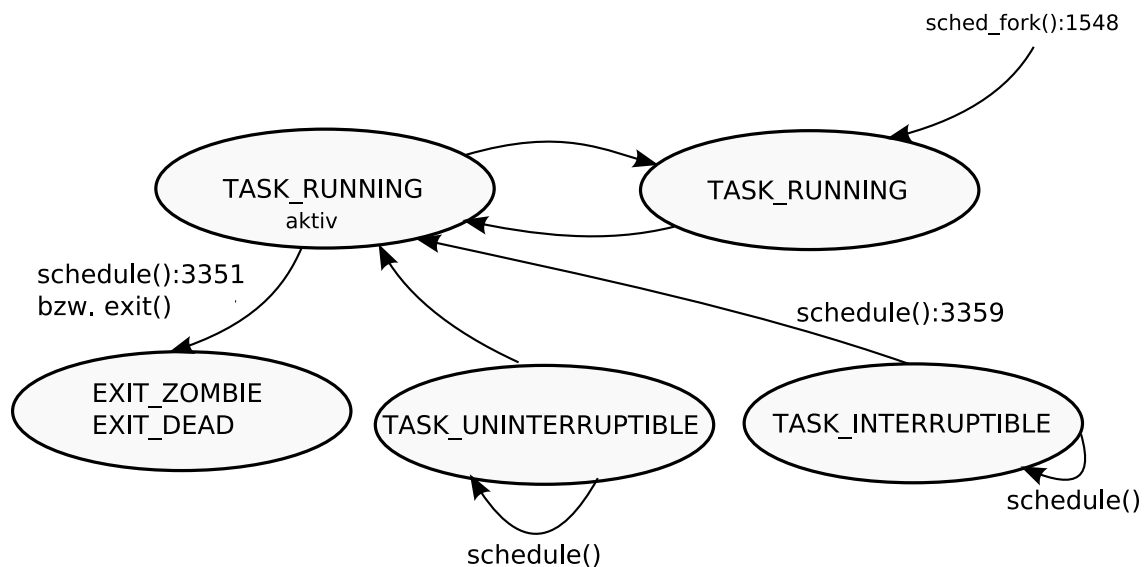
- *Nach dem Theorem von Nyquist können unbegrenzte Datenraten in der Praxis erreicht werden.*

Nein. Die Formel von Nyquist lässt nämlich auftretendes Rauschen in der Leitung völlig außen vor, geht also von keinem Rauschen aus. In der Praxis gibt es in den Leitungen aber immer einen Störfaktor, so dass man die Anzahl der Symbole also nicht beliebig hochsetzen kann, um die Datenrate unbegrenzt ansteigen zu lassen.

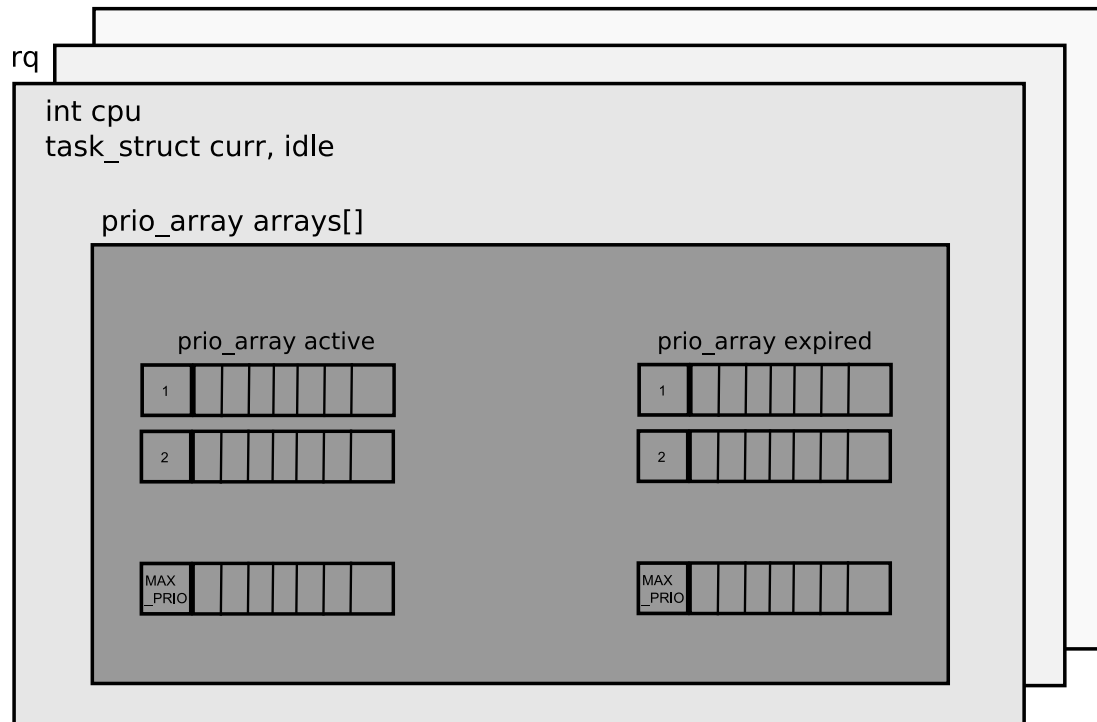
- *Um die maximal mögliche Datenrate zu ermitteln, muß man das Maximum der nach den Formel von Nyquist und Shannon berechneten Werte nehmen.*

Nein. Die maximale Datenrate wird natürlich bei gegebener Signalstärke und Störpegel, Bandbreite und Symbolanzahl durch das kleinere Ergebnis der beiden Formeln begrenzt.

5. Scheduler



So wirklih sicher bin ich mir nicht. Irgendwie muss man ja auch zu *TASK_INTERRUPTIBLE* und so kommen, aber das habe ich im Quellcode nicht gefunden. War alles sehr merkwürdig.



Pro CPU gibt es eine sogenannte "runqueue". Hauptsächlich enthält diese Struktur ein Prioritätsarray (*prio_array arrays*), in dem zwei Prioritätswarteschlangen-Arrays verwaltet werden, nämlich *prio_array active* und *prio_array expired*. Dabei bestehen diese beiden Warteschlangen-Strukturen aus jeweils *MAX_PRIO* Warteschlangen, in denen jeweils sich die Tasks mit der entsprechenden Priorität befinden. In *active* sind dabei solche Prozesse, deren Zeitfenster noch nicht aufgebraucht ist und, in *expired* folglich solche, deren Zeitfenster bereits verbraucht ist.

6. Webserver

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/socket.h>
4 #include <sys/types.h>
5 #include <netdb.h>
6 #include <netinet/in.h>
7 #include <unistd.h>
8 #include <string.h>
9
10 #define BUFFER 512
11 #define TRUE 1
12
13 //Findet den Dateityp raus
14 char* findeTyp(char *pfad) {
```

```
15     if(strstr(pfad, ".html") != 0)
16         return "text/html";
17     else if(strstr(pfad, ".jpg") != 0)
18         return "image/jpeg";
19     else if(strstr(pfad, ".gif") != 0)
20         return "image/gif";
21     else
22         return "text/plain";
23 }
24
25 // Behandelt das lesen und schreiben vom Socket
26 int lesenSchreiben(int sockRem, char *verz) {
27     FILE *read, *write, *dat;
28     char bufRead[BUFFER], bufWrite[BUFFER];
29     char *datei[2], header[200], *pfad, *tmp, *typ;
30
31     // Socket zum lesen und schreiben als Datei öffnen (
32     // damit keine Fehler kommen)
33     read = fdopen(sockRem, "rb");
34     write = fdopen(sockRem, "wb");
35     setvbuf(read, bufRead, _IOLBF, sizeof(bufRead));
36     setvbuf(write, bufWrite, _IOLBF, sizeof(bufWrite));
37
38     printf("Sockets wurden zum Lesen und Schreiben geöffnet\n");
39
40     tmp = malloc(sizeof(verz));
41     strcpy(tmp, verz);
42
43     if(fgets(bufRead, sizeof(bufRead), read) == NULL) {
44         perror("Fehler mit dem Einlesen\n");
45         return -1;
46     }
47
48     datei[0] = strtok(bufRead, "\n"); // Bei richtiger
49     // Anfrage GET
50     datei[1] = strtok(NULL, "\n"); // Angefragte Datei
51     if(strcmp(datei[0], "GET") != 0) {
52         perror("Fehler mit GET\n");
53         return -1;
54     }
55
56     printf("Die GET Anfrage wurde gelesen, Datei %s wurde\n", datei[1]);
57
58     // Index.html angefragt?
59     if(strcmp(datei[1], "/") == 0 || strcmp(datei[1], "/index.html") == 0)
```

```
56         pfad = strcat(verz, "/index.html");
57     else // nee, ne andere Datei
58         pfad = strcat(verz, datei[1]);
59
60     // Dateityp rausfinden
61     typ = findeTyp(pfad);
62     printf("Typ: %s\n", typ);
63
64     // Datei vorhanden ??
65     if (access(pfad, R_OK) == 0) {
66         printf("Pfad %s ist vorhanden\n", pfad);
67         sprintf(header, "HTTP/1.1 200 OK\r\nContent -
68             Type: %s\r\n\n", typ);
69         dat = fopen(pfad, "rb");
70         fprintf(write, "%s", header); // schreibt Header
71                                     // in Socket-Datei
72         printf("Folgendes wird an den Client gesendet:\n%s", header);
73         while (!feof(dat)) {
74             memset(bufWrite, 0, sizeof(bufWrite)); // mit
75                                                     // 0 vollschreiben
76             fread(bufWrite, 1, sizeof(bufWrite), dat);
77             fwrite(bufWrite, 1, sizeof(bufWrite), write);
78             printf("%s", bufWrite);
79         }
80         fprintf(write, "\n"); // Irgendwie braucht er zum
81                               // Schluss nen Zeilenumbruch ...
82         fclose(dat); // Datei schließen
83     }
84     else { // Datei nicht gefunden, dann 404 Meldung
85         printf("Pfad %s nicht vorhanden, deshalb 404
86             Meldung\n", pfad);
87         sprintf(header, "HTTP/1.1 404 Not Found\r\n\n<
88             html>\n<head><title>Fehler 404</title></head>
89             \n<body>ERROR 404 - File not found.</body>\n
90             </html>\n");
91         fprintf(write, "%s", header); // schreibt in den
92                                     // Socket
93         printf("Folgendes wird an den Client gesendet:\n%s", header);
94     }
95     printf("\nErfolgreich geschrieben\n");
96     // Alles aufräumen und so
97     strcpy(verz, tmp); // Damit wir beim nächsten Aufruf
98                         // immer noch im selben Verzeichnis gucken
99     free(tmp);
```

```
90         fclose(read);
91         fclose(write);
92         return 0;
93     }
94
95     int my_webserver(int port, char *verzeichnis) {
96         int sock, sockRem, client_size;
97         struct sockaddr_in server, client;
98
99         sock = socket(AF_INET, SOCK_STREAM, 0);
100        if(sock == -1) {
101            perror("Fehler_mit_Socket\n");
102            return -1;
103        }
104
105        server.sin_addr.s_addr = INADDR_ANY;
106        server.sin_port = htons(port);
107        server.sin_family = AF_INET;
108
109        if(bind(sock, (struct sockaddr *) &server, sizeof(
110            server)) == -1) {
111            perror("Fehler_bei_bind\n");
112            return -1;
113        }
114        if(listen(sock, 1) == -1) {
115            perror("Fehler_bei_listen\n");
116            return -1;
117        }
118        while (TRUE) { // immer lauschen
119            printf("Warten_auf_Anfrage...\n");
120            client_size = sizeof(client);
121            sockRem = accept(sock, (struct sockaddr *) &
122                client, &client_size);
123            if(sockRem == -1) {
124                perror("fehler_mit_accept\n");
125                return -1;
126            }
127            else
128                lesenSchreiben(sockRem, verzeichnis);
129            close(sockRem);
130            printf("Anfrage_bearbeitet._Clientsocket_
131                geschlossen\n");
132        }
133        close(sock); // Sollte er aber nie erreichen
134        return 0;
```

```
133 }
134
135 int main(int argc, char* argv[]) {
136     int port;
137     if (argc != 3)
138         fputs("Fehler, zuwenig/zuviel Parameter",
139             stderr);
139     else {
140         port = atoi(argv[1]);
141         printf("Server auf Port %i und im Verzeichnis %s
142             geöffnet.\n", port, argv[2]);
143         my_webserver(port, argv[2]);
144     }
145     return EXIT_SUCCESS;
146 }
```