No signs ? No problem !

Zakaria Kasmi

1 Einführung

Alle numerischen Datentypen sind in Java vorzeichenbehaftet. Die bekannten vorzeichenlosen Datentypen: **unsigned char**, **unsigned short**, **unsigned int** und **unsigned long** wie in C und C++ gibt es in Java nicht.

Alle primitiven Datentypen haben in Java eine feste Speichergröße und Genauigkeit, da sie plattformunabhängig sind. Dagegen sind die Speichergröße und die Genauigkeit der numerischen Datentypen in C oder C++ von der Zielplattform abhängig. In der unteren Tabelle 1 findet man mögliche primitive Daten in Java und einen Vergleich zwischen den Java primitiven Datentypen und den C Datentypen.

Type	Java	C
char	16 Bits (Unicodes)	8 Bits (ASCII)
short	16 Bits	16 Bits
int	32 Bits	16, 32, oder 64 Bits
long	64 Bits	32 oder 64 Bits
float	32 Bits	32 Bits
double	64 Bits	64 Bits
boolean	1 Bit	-
byte	8 Bits	s. char
long long	-	64 Bits (inoffiziell)
long double	-	80, 96 oder 128 Bits

Tabelle 1: Vergleich zwischen den Java und C-Datentypen. Unterstrichen sind die heute gebräuchlichen Speichergrößen

Jeder primitiver Datentyp in **Java** gehört zu einer korrespondierenden **Wrapper-Klasse**, die die primitive Variable in einer objektorientierten Hülle kapselt. Z.B. für den Typ **double** gibt es die Klasse **Double**, die eine Reihe von Methoden zum Zugriff auf diese primitive Variable bereitstellt.

Obwohl es eine große Anzahl von Netzwerkprotokollen, Dateiformaten und Hardware-Schnittstellen gibt, die die vorzeichenlosen Datentypen benutzen, werden sie trotzdem von Java nicht unterstützt. Die einzige Ausnahme ist der Datentyp **char**, der ein Unicode-Zeichen mit der Länge 16 bit darstellt. Anstelle von dem C-„**char**", der ein Byte-ASCII-Zeichen repräsentiert.

2 Wieso unterstützt Java keine vorzeichenlosen Datentypen?

Obwohl es eine Menge von Netzwerkprotokollen gibt, die diese Datentypen benutzen, fragt man sich wieso bietet Java keine vorzeichenlose Zahlen?

In einem interessanten Interview mit **James Gosling,** einer der Erfinder der Programmiersprache Java, antwortet er indirekt auf diese Frage. Der Artikel ist in der Zeitschrift „Java Report, 5(7), July 2000" und „C++ Report, 12 (7), July/August 2000" erschienen, in dem Dennis Ritchie, Bjarne Stroustrup und James Gosling die Erfinder von C, C++ und Java, interviewt werden.

Ein Ausschnitt aus diesem Interview:

> ***Q:...***
>
> ***Q: Programmers often talk about the advantages and disadvantages of programming in a "simple language." What does that phrase mean to you, and is [C/C++/Java] a simple language in your view?***
>
> **Ritchie:** [wird weggelassen]
>
> **Stroustrup:** [wird weggelassen]
>
> **Gosling:** For me as a language designer, which I don't really count myself as these days, what "simple" really ended up meaning was could I expect J. Random Developer to hold the spec in his head. That definition says that, for instance, Java isn't -- and in fact a lot of these languages end up with a lot of corner cases, things that nobody really understands. Quiz any C developer about unsigned, and pretty soon you discover that almost no C developers actually understand what goes on with unsigned, what **unsigned arithmetic** is. Things like that made C complex. The language part of Java is, I think, pretty simple. The libraries you have to look up.

Aus der Antwort von Gosling kann man schließen, dass man vorzeichenlose Datentypen aus den folgenden Gründen nicht eingeführt hat:

- Man möchte Java so einfach wie möglich halten.
- Die Komplexität der vorzeichenlosen Arithmetik dem Entwickler ersparen.
- Die virtuelle Maschine einfach halten, da mehr Datentypen eine komplexere und langsamere virtuelle Maschine bedeutet.

3 Lösung des Problems der vorzeichenlosen Datentypen

Bevor wir der Problemlösung widmen, betrachten wir das Bitformat von vorzeichenbehafteten und vorzeichenlosen Datentypen sowie auch eine Byte-orientierte Kommunikation zwischen zwei Transceivern, die mit Hilfe von Datenpaketen Daten tauschen.

Datentyp	Größe	Wertebereich
char, **signed char**	1 Byte = 8 Bit	-128 … +127
unsigned char	1 Byte = 8 Bit	0 … +255
int, **signed integer**	2 Byte = 16 Bit	-32768 … +32767
unsigned int	2 Byte = 16 Bitt	0 … +65535

Tabelle 2: Einige Datentypen in der C-Sprache mit Wertebereich

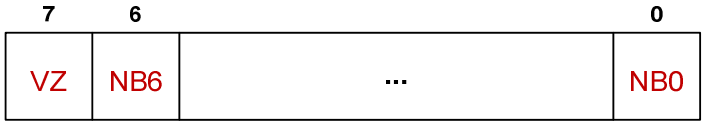

Abbildung 1: Bitformat eines vorzeichenbehafteten Datentyps (signed char)

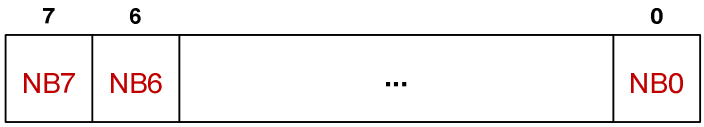

Abbildung 2: Bitformat eines vorzeichenlosen Datentyps (unsigned char)

NB_i: Nutzbit mit der Wertigkeit i
VZ: Vorzeichenbit

Wie aus den Abbildungen 1-2 und aus der Tabelle 2 zu erkennen ist, wird das höchstwertige Bit (MSB[1] = most significant bit) bei vorzeichenbehafteten Datentypen als Vorzeichen-Bit reserviert. Deshalb hat ein **char** nur 7 Nutzbits und sein Wertebereich ist: $-2^7 \ldots 0 \ldots 2^7-1$ das entspricht -128…0...127. Dagegen gibt es bei den vorzeichenlosen Datentypen kein Vorzeichenbit und sie haben eine volle Nutzbitanzahl. Ein **unsigned char** hat 8 Nutzbits und sein Wertebereich ist: $0 \ldots 2^8-1$ das entspricht 0…255. Die Größe eines Datentyps in Byte kann in C über den Operator „**sizeof**" sowie seinen Wertebereich über die Makros, die in der Headerdatei „**limit.h**" definiert sind, ermittelt werden. Als Beispiel geben die beiden „**INT_MIN**"- und „**INT_MAX**"-Makros den Wertebereich eines Integerdatentyps aus.

Um die Problematik des vorzeichenlosen Datentypen zu verdeutlichen, betrachten wir den Fall, in dem zwei Kommunikationsinstanzen (Client/Server) Byte-orientiert Daten tauschen. Das Client- wurde in Java, dagegen das Server-Programm in C implementiert (Vergleiche die Abbildung 3). In diesem Fall wird ein Bytestrom (Stream), der einige vorzeichenlose Zahlen enthält, aus der Netzwerks- gelesen bzw. in die Netzwerksschnittstelle geschrieben.

[1] MSB steht für Most Significant Bit. Es ist das Bit mit der höchsten Wertigkeit.

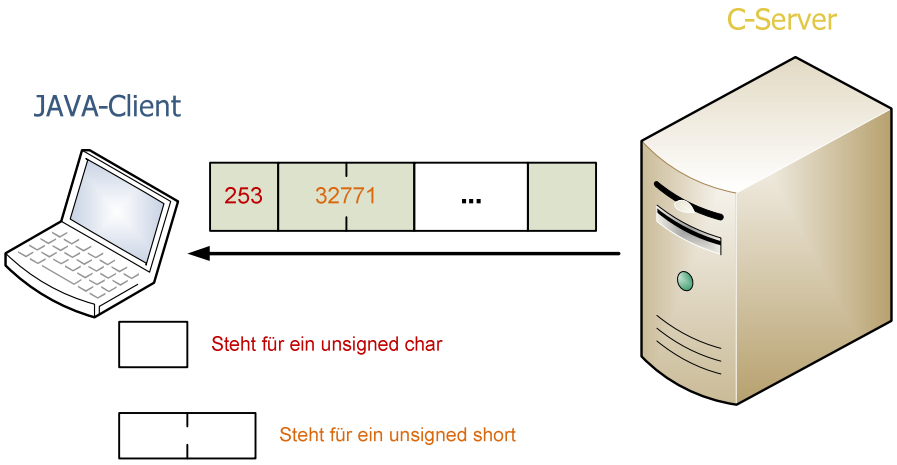

Abbildung 3: C-Server sendet ein "unsigned char"- und einem "unsigned short"-Datentyp zu einer Java-Applikation

Wie aus der Abbildung 3 zu erkennen ist, empfängt das Java-Client ein vorzeichenloses char ($z1_c$ = 253) und ein vorzeichenloses short ($z2_c$ = 32771), die von der Java-Client-Applikation falsch interpretiert werden. Sie wird die Zahlen als $z1_j$ = -3 und $z2_j$ = -32765 interpretieren. Angesichts der Tatsache, dass negative Zahlen sowohl in der C- als auch in der Java-Sprache im Zweierkomplementformat kodiert werden, kann die falsche Interpretation wie folgt veranschaulicht werden:

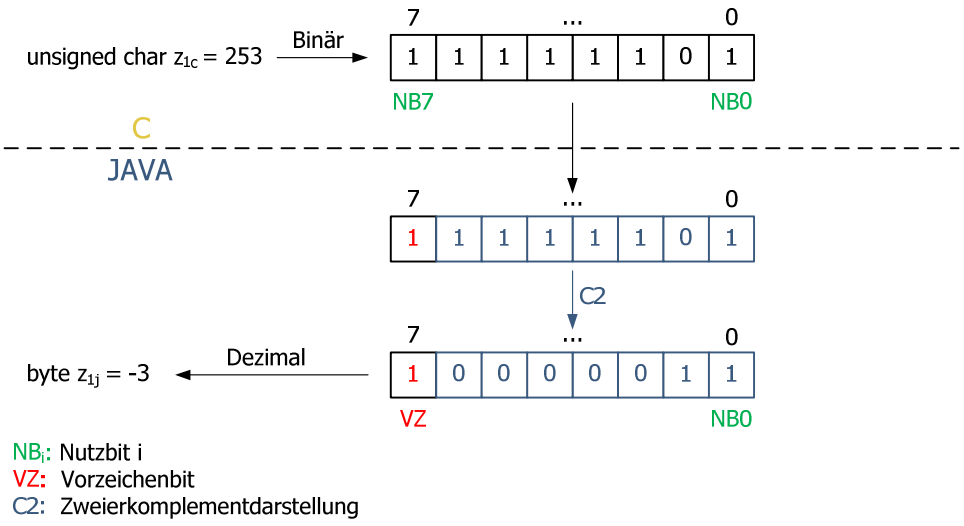

Abbildung 4: Falsche Interpretation eines "unsigned char"

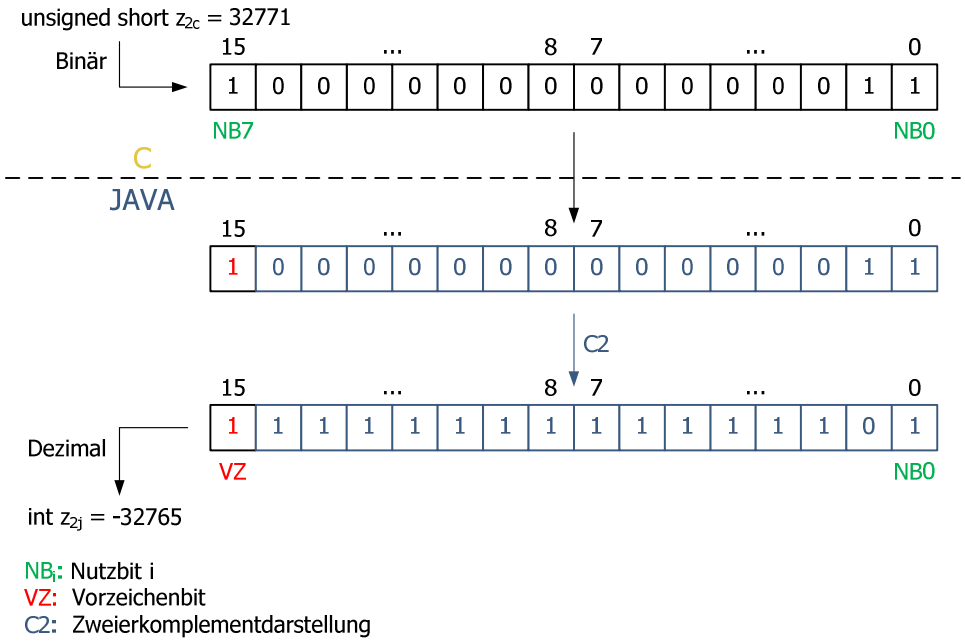

Abbildung 5: Falsche Interpretation eines "unsigned short"

Aus dem vorher betrachteten Bitformat und den beiden Beispielen in den Abbildungen 4 und 5 erkennt man, dass das auf 1 gesetzte MSB-Bit (höchstwertiger Nutzbit) eines vorzeichenlosen Datentyps von der Java-Applikation als Vorzeichenbit erkannt wird. In dieser Weise wird die ursprüngliche positive Zahl als eine negative Zahl von der Java-Anwendung gespeichert. In der Abbildung 5 wird der vorzeichenlose Datentyp „**unsigned short**" als eine negative Zahl dargestellt, obwohl der Datenbereich eines Java-Integers doppelt so groß als eines „**unsigned short**"-Datentyps in C ist. Der Grund liegt an dem Vorzeichenerweiterungsproblem (engl.: sign extension problem), bei dem die ursprüngliche Zahl erweitert wird, um die neue Zahl (Zielformat) mit den Vorzeichenbits zu füllen.

Der allgemeine Ansatz ist, dass man den nächsten größeren vorzeichenbehafteten Datentyp verwendet, um den ursprünglichen vorzeichenlosen Datentyp zu speichern. Man benutzt ein **short**, um einen **vorzeichenlosen byte** zu speichern, ein **char**, um einen **vorzeichenlosen short** zu speichern und ein **long**, um einen **vozeichenlosen int**, zu speichern. Dazu muss man das Vorzeichenerweiterungsproblem vermeiden. Im Falle einer Client/Server-Architektur muss man einen Strom (Stream) von Bytes, der einige vorzeichenlose Zahlen enthält, aus dem Netzwerk lesen bzw. ins Netzwerk schreiben. Damit die Daten richtig in die nächsten größeren Datentypen konvertiert werden, muss man etwas mehr Aufwand betreiben. Im Folgenden wird eine Lösung für das Lesen und Schreiben aus einer Netzwerkschnittstelle vorgeschlagen:

A Lesen aus der Netzwerksschnittstelle

Lesen eines vorzeichenlosen Bytes

Ein Byte wird durch die folgende Operation in ein vorzeichenloses Byte konvertiert:

```
byte signedByte;

short unsignedByte = (short) ((int) signedByte & 0xFF);
```

Jede Zahl, deren Wert größer als 127 ist, wird von Java als eine negative Ziffer betrachtet. Deshalb wird die Umwandlung in zwei Schritten ausgeführt:

1. Die Zahl wird in ein Integer gecastet[2], dabei werden die Bits 0 bis 7 die gleichen wie ein Byte sein und die Bits 8 bis 31 werden auf 1 gesetzt.
2. Durch die UND-Verknüpfung mit **0x000000FF** werden die Bits von 8 bis 31 mit 0 und dadurch das Vorzeichen-Bit (31. Bit) initialisiert.

Und so wird die Zahl von Java als positiv interpretiert. Eine kompakte Darstellung von **0x000000FF** ist **0xFF**.

Da die Konvertierung analog wie bei den vorzeichenlosen Bytes funktioniert, werden die nächsten vorzeichenlosen Datentypen nicht ausführlich behandelt.

Lesen eines vorzeichenlosen Short

Ein Short wird durch die folgende Operation in ein vorzeichenloses Short konvertiert:

```
short signedShort;

char unsignedShort = (char) ((int) signedShort & 0xFFFF);
```

Lesen eines vorzeichenlosen Integer

Ein Integer wird durch die folgende Operation in ein vorzeichenloses Integer konvertiert:

```
int signedInteger;

long unsignedInteger = (long) (( signedInteger & 0xFFFFFFFFL);
```

Damit Java die Zahl richtig interpretiert muss ein **L** zu **0xFFFFFFFF** angehängt werden. Sonst kehrt man wieder zu dem Vorzeichenerweiterungsproblem (engl.: **sign extension problem**) zurück.

B Schreiben in die Netzwerksschnittstelle

Die oben definierten Zahlen werden byteweise in einen Puffer zwischengespeichert und dann über das Netz gesendet.

Schreiben eines vorzeichenlosen Integer

Die Zahl wird in vier Bytes geteilt und in ein Array gepuffert. Jedes Byte wird in zwei Schritten in den Puffer gespeichert:

1. Eine bitweise **UND**-Verknüpfung.
2. Eine Rechtsverschiebung um y Stellen, wobei y/8 ist die Index des Puffers ist.

[2] Deutsch: Typeumwandlung d.h. die Umwandlung des Wertes eines Datentyps in einen Wert eines anderen Datentyps.

```
byte[] buff = new byte[4];

            buff[3] = (byte) ((unsignedInteger & 0xFF000000L) >> 24);

            buff[2] = (byte) ((unsignedInteger & 0x00FF0000L) >> 16);

            buff[1] = (byte) ((unsignedInteger & 0x0000FF00L) >>  8);

            buff[0] = (byte) ((unsignedInteger & 0x000000FFL)      );
```

Schreiben eines vorzeichenlosen Short

Die Zahl wird in zwei Bytes geteilt und in ein Array gepuffert:

```
byte[] buff = new byte[2];

            buff[1] =  (byte) ((unsignedShort & 0xFF00) >> 8);

            buff[0] =  (byte) (unsignedShort & 0x00FF);
```

Schreiben eines vorzeichenlosen Bytes

```
byte b[] = {(byte)(unsignedByte & 0xFF)};
```

Man kann die neuen Datentypen in eigene Klassen die sog. **Wrapper-Klassen** definieren. Zu jedem primitiven vorzeichenlosen Datentyp gibt es eine korrespondierende Wrapper-Klasse. Diese kapselt die primitive Variable in einer objektorientierten Hülle und stellt eine Reihe von Methoden zum Zugriff auf die Variable zur Verfügung.

Folgende Wrapper-Klassen können definiert werden:

- **UnsignedByte:** verhüllt einen Wert des primitiven vorzeichenlosen **byte**.
- **UnsignedShort:** verhüllt einen Wert des primitiven vorzeichenlosen **short**.
- **UnsignedInteger:** verhüllt einen Wert des primitiven vorzeichenlosen **integer**.

Beim Zugreifen auf die vordefinierten vorzeichenlosen Datentypen kann nicht immer garantiert werden, dass der Zugriff immer atomar ist, falls mehrere Threads[3] gleichzeitig auf den Datentyp zugreifen. Das Problem kann durch die Thread-Synchronisation bzw. durch das Monitor-Konzept gelöst werden.

[3] Ein Thread (engl. für *Faden*) beschreibt eine nebenläufige Ausführungseinheit innerhalb eines einzigen Prozesses. Mehrere Threads können parallel innerhalb eines Prozesses laufen und auf dieselben Objekte zugreifen.

4 Flexible Paketdefinition und Zugriff auf die Netzwerkschnittstelle

Die Datenpakete (abgekürzt Pakete) spielen bei der Kommunikation eine entscheidende Rolle, da die Kommunikationspartner (z.B. Transceiver) mit deren Hilfe kommunizieren. Sie sind die Sprache der Transceiver. Ein Datenpaket besteht aus mehreren Datenfeldern, wobei sich jedes Datenfeld aus mehreren Bytes zusammen setzt (Vergleiche Abbildung 6). Ein Paket besteht meistens aus zwei Teilen:

1. Header (Kopfdaten): enthalten Informationen über den Pakettyp, Identifikationsnummer einer Nachricht, Prüfsumme (CRC: Cyclic Redundancy Check), usw.
2. Nutzdaten: die eigentlichen Daten (Parameter) wie die empfangene Signalleistung und die empfangene Rauschleistung.

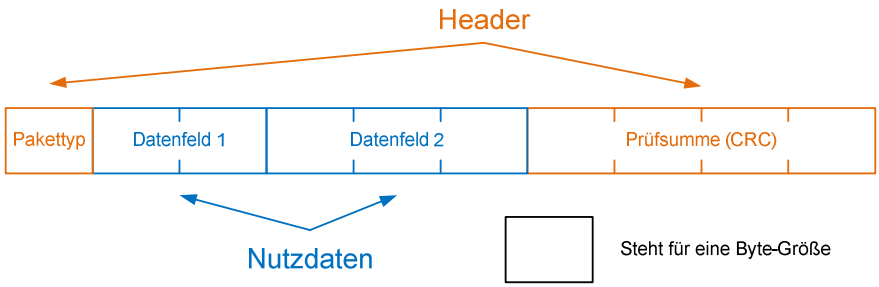

Abbildung 6: Allgemeines Format eines Datenpakets

Es gibt in Java keine **„memcpy"-Funktion** wie in C oder in C++, die einen beliebigen Speicherbereich Byte für Byte kopiert. Der Programmierer muss selber die Bytes in der richtigen Reihenfolge und Größe in die Netzwerksschnittstelle schreiben bzw. aus der Netzwerksschnittstelle lesen. In C bzw. in C++ hätte man jedes **Paket** als ein Datenverbund **(Struct)** definiert und bei einem Lesezugriff einfach die Bytes mit Hilfe der **„memcpy"-Funktion** in den Datenverbund kopiert. Beim Schreiben können die Daten mit Hilfe der gleichen Funktion vom **„Struct"** in einen Bytes-Array zwischengespeichert werden und der Netzwerkschnittstelle übergeben werden.

In Java werden die Pakete in Klassen definiert. Die Datenfelder entsprechen die Attribute und die Methoden erlauben das Lesen aus einem Ausgabestrom bzw. das Schreiben in einen Ausgabestrom ohne jedes Byte analysieren zu müssen. Damit das Programm flexibel gehalten wird, kann jedes Paket neu definiert oder aus dem Ausgabestrom gelesen bzw. in den Eingabestrom geschrieben werden ohne jedes Byte einzeln auszuwerten. Erstmals kann man eine Schnittstelle mit dem Namen Packet definieren, die von der Klasse **Default-Paket** implementiert wird. In der Klasse **Default-Paket**, die von allen **Paketen** geerbt wird, wurden die folgenden Funktionen definiert:

- **setPacketFields**(**byte**[] buf): initialisiert die **Paketfelder** mit den entsprechenden Bytes aus der Netzwerk-Schnittstelle.
- **byte**[] **toBytes**(**int** packetLength): konvertiert jedes **Paket** in Bytes.
- **int getOffset**(**Obj** obj): gibt ein Offset bzw. die Anzahl der Bytes für einen Paketfeld an.
- **Packet bytesToPacket(byte[]** bArr**)**: bildet aus mehreren Bytes ein Paket.

Die Funktion **toBytes()** läuft jedes Paketfeld durch und mit Hilfe der **getOffset()-Funktion** werden für jeden Paketeintrag die entsprechenden Bytes in einem Puffer reserviert. Am Ende wird der gesamte Puffer, der die Paketfelder als Bytes speichert, zur Netzwerk-Schnittstelle übergeben. Die Funktion **setPacketFields()** funktioniert nach dem gleichen Prinzip. Die untere Abbildung 7 schildert die Funktionsweise der **toBytes()-Funktion**.

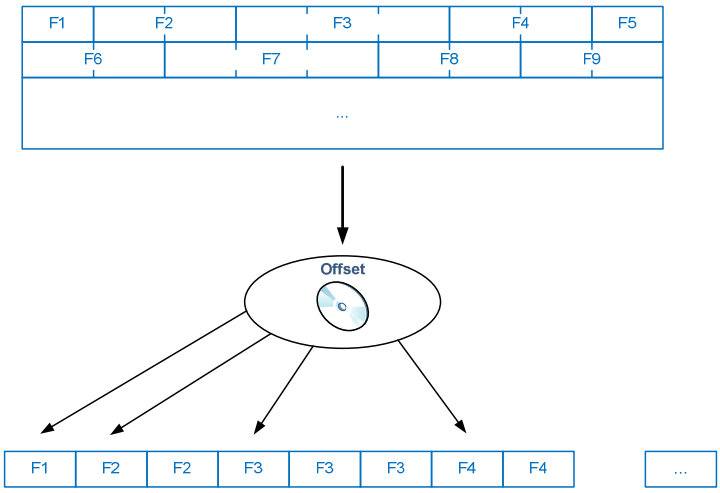

Abbildung 7: Funktionsweise der toBytes()-Funktion

5 Implementierung eines CRC-Algorithmus

Das CRC-Verfahren stellt eine effiziente Methode zur Fehlererkennung dar und wird als CRC (Cyclic Redundancy Code) abgekürzt. Es gehört zu den Polynomecodes. CRC hat die folgenden Eigenschaften:

- Das Overhead ist von der Kodewortlänge unabhängig.
- Geringer Rechenaufwand beim Sender und Empfänger.
- Sehr gute Fehlererkennungs-Wahrscheinlichkeit von Mehrfachfehlern bzw. Bündelfehlern (bursty errors).

In Wirklichkeit treten die Fehler in Bündeln aufgrund der physikalischen Eigenschaften wie elektrische Störungen, der Verlust der Bitsynchronisation und das Nebensprechen zwischen den Übertragungskanälen auf. Bündelfehler sind schwer zu erkennen und zu modellieren, was ein weiterer Grund ist, um eine CRC- zusätzlich zur Internet-Prüfsumme zu benutzen.

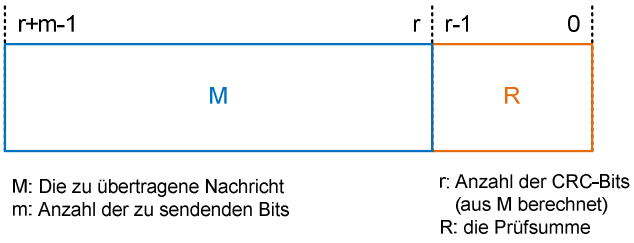

Abbildung 8: Eine Nachricht mit CRC-Bits

Wie aus der Abbildung 8 ersichtlich ist, wird eine Gruppe R von r Fehlerkontrollbits zu der Nachricht M, die zu übertragen ist, addiert. Diese Sicherungssequenz wird manchmal auch als **FCS: Frame Check Sequence** bezeichnet.

Da die Transceiver einen vorzeichenlosen Integerwert als Prüfsumme berechnen, wurde eine eigene Klasse anstatt der CRC32-Klasse von Java implementiert. Für die Berechnung wird eine CRC-Tabelle, die 32-Hashwerte enthält, verwendet. Die Prüfsumme wird berechnet, indem jedes Daten-Byte mit einem Hashwert aus der Tabelle verknüpft wird. Die CRC-Tabelle ist für die beiden Transceiver gemeinsam.

Hiermit ist ein Ausschnitt aus der Tabelle:

```
crcTable[] = {

        0x0,
        0x04c11db7, 0x09823b6e, 0x0d4326d9, 0x130476dc,
        0x17c56b6b, 0x1a864db2, 0x1e475005, 0x2608edb8, 0x22c9f00f,
        0x2f8ad6d6, 0x2b4bcb61, 0x350c9b64, ….

}
```

In Abbildung 9 wird der Algorithmus für die CRC-Prüfsumme skizziert. Der Algorithmus funktioniert wie folgt:

1. CRC-Wert wird mit **0xFFFFFFFFL** initialisiert.
2. Das höchstwertige Byte des CRC-Wertes wird mit einem Wert aus der CRC-Tabelle XOR- verknüpft
3. Ein Index wird aus der XOR-Verknüpfung zwischen den ersten Bytes des CRC-Wertes berechnet. Mit diesem Index wird ein Wert für den 1. Schritt aus der Tabelle gelesen.
4. Wiederhole Schritt 2 und Schritt 3 für jedes Datenbyte.

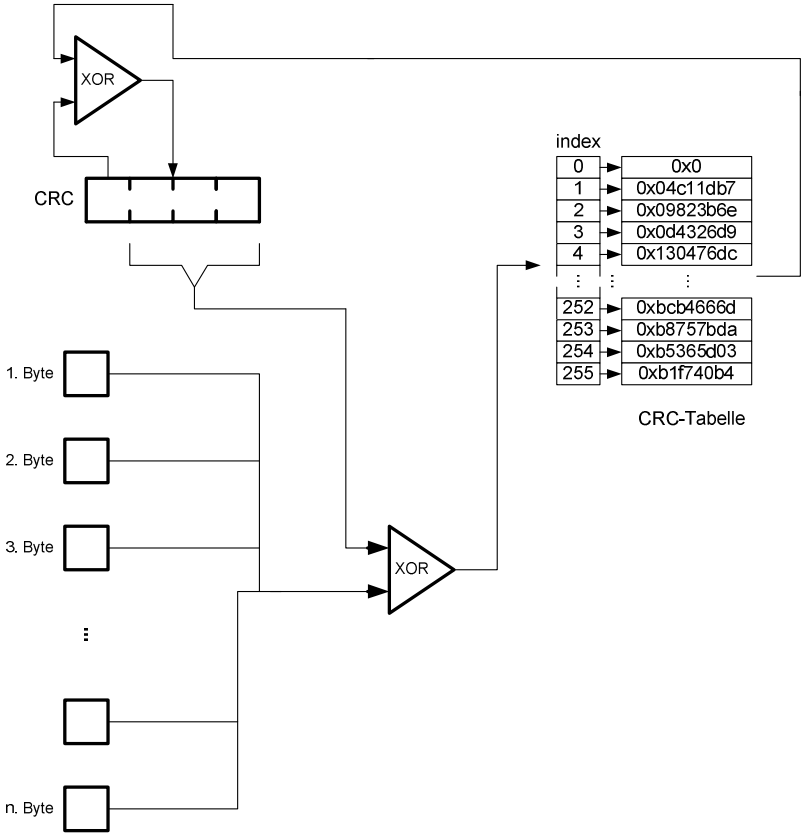

Abbildung 9: CRC-Algorithmus (Software-Implementierung)