Programmieren C: Bitoperationen: Reflektierter Gray-Code

Klaus Kusche

Ein ***Gray-Code*** ist eine Zuordnung von *Bitmustern* zu ganzen Zahlen,
der sich von der normalen Binärdarstellung ganzer Zahlen dadurch unterscheidet,
dass sich ***im Code aufeinanderfolgender Zahlen immer nur genau ein Bit ändert***.

Es gibt *viele Möglichkeiten*, solche Gray-Codes zu definieren,
Wir beschränken uns hier auf Codes, die *bei n Bits 2^n verschiedene Werte* enthalten.
Die *häufigste und einfachste Variante* ist der sogenannte ***reflektierte Gray-Code***.
Er heißt deshalb so, weil bei einem n-Bit-Code die *hinteren n-1 Bits*
der oberen und der unteren Hälfte der Werte *genau spiegelbildlich zueinander* sind
(bzw. ganz allgemein gilt: Wenn man die 2^n Codes in Blöcke von jeweils 2^i Codes aufteilt,
dann sind die hinteren i Bits der Codes im ersten und im zweiten solchen Block
spiegelbildlich zueinander).

Eine andere, *kompliziertere Möglichkeit* ist der sogenannte ***balanzierte Gray-Code***
(mit dem wir uns hier nicht befassen). Bei ihm lautet die Konstruktionsvorschrift,
dass *jedes Bit gleich oft den Zustand wechselt*, also z.B. bei einem 4 Bit Code 4 mal.
(insgesamt 16 Werte, daher 16 Bitwechsel, verteilt auf 4 Bits ergibt 4 Wechsel/Bit).

Gray-Codes mit 4 Bits sehen beispielsweise so aus (links reflektiert, rechts balanziert):

0	**0000**	**0000**
1	**0001**	**0001**
2	**0011**	**0011**
3	**0010**	**1011**
4	**0110**	**1111**
5	**0111**	**0111**
6	**0101**	**0101**
7	**0100**	**0100**
8	**1100**	**0110**
9	**1101**	**0010**
10	**1111**	**1010**
11	**1110**	**1110**
12	**1010**	**1100**
13	**1011**	**1101**
14	**1001**	**1001**
15	**1000**	**1000**

Anwendung finden Gray-Codes unter anderem in der *parallelen Signalübertragung*:

Überträgt man beispielsweise *n bit breite Messwerte* von Sensoren *über n Leitungen*
als normale Binärzahlen, und der Messwert ändert sich um 1, dann kann der Empfänger
völlig falsche Werte lesen, wenn er die Leitungen *genau im Umschalt-Zeitpunkt* abfragt
und nicht alle n Leitungen exakt zum selben Zeitpunkt umschalten.

Extremes Beispiel: 8-Bit Werte, und der Messwert ändert sich von **127** auf **128**,
d.h. in *normaler Binär-Codierung* von **01111111** auf **10000000**. Erfolgt die Umschaltung
einer Leitung von **0** auf **1** etwas *langsamer* als die von **1** auf **0**, so sieht der Empfänger
im Umschaltzeitpunkt kurzzeitig **00000000**, also **0**. Ist die Umschaltung von **0** auf **1**
etwas *schneller*, kommt kurzzeitig **11111111** beim Empfänger an, also **255**.

Bei einer Übertragung mittels Gray-Code ändert sich nur eine einzige Leitung,
wenn sich der Messwert um 1 ändert, und egal wie schnell sich die Leitung ändert
und wann der Empfänger liest: Der Empfänger sieht *entweder die alte oder die neue Zahl*,
aber *nie eine davon abweichede Zahl*.
Außerdem minimiert der Gray-Code die Anzahl der Spannungswechsel und damit
(insbesondere bei CMOS-Schaltungen) den *Stromverbrauch* der Übertragung.

Auch in der *Fehlerkorrektur von QAM-Modulationsverfahren* spielen Gray-Codes eine Rolle,
ebenso in *einigen mathematischen und algorithmischen Problemstellungen*.

a) Hauptprogramm und allgemeine Hilfsfunktionen:

Gesucht ist ein Programm, das mit einer *positiven ganzen Zahl* **`n`** (der *Anzahl der Bits*)
auf der Befehlszeile aufgerufen wird und eine *Tabelle mit dem reflektierten Gray-Code*
mit **`n`** Bits ausgibt.

Das *Hauptprogramm und auch alle Funktionen* sollen ***ohne*** *Multiplikation, Division und Restrechnung* und auch ***ohne*** **`pow`** auskommen, verwende stattdessen *Bitoperationen*.
Alle ganzen Zahlen sind *vorzeichenlos*, definiere dafür **`uint`** als *Typ-Abkürzung*.

- Zuerst brauchen wir eine **Hilfsfunktion**, die eine *Zahl binär ausgibt*.
  Diese Funktion bekommt *zwei ganze Zahlen* als Parameter:
  Die *auszugebende Zahl* und die *Anzahl* **`n`** *der Bits*, die ausgegeben werden soll
  (wir meinen immer die *hintersten* **`n`** *Bits*). Returnwert hat die Funktion keinen.

  Die Funktion soll *Bit für Bit einzeln* ausgeben (mit **`putchar`**, nicht **`printf`**),
  ohne Zwischenräume oder Zeilenvorschübe.

  Ich empfehle die schon geübte Vorgangsweise: Auszugebendes Bit
  ganz nach hinten schieben, alles andere *wegmaskieren*, auf ASCII-Code *umrechnen*.
- Um die *Korrektheit* unserer berechneten Gray-Codes zu prüfen, brauchen wir
  eine **Hilfsfunktion**, die *zu einem Gray-Code wieder die ursprüngliche Zahl* liefert
  und daher den *Gray-Code als Parameter* hat
  und die *korrespondierende Binärzahl als Returnwert* liefert.

  Die Funktion arbeitet wie folgt:

  - *Initialisiere* das Ergebnis mit **`0`**.
  - Wiederhole in einer *Schleife* Folgendes, bis der Gray-Code **`0`** geworden ist:
    - *Invertiere* im Ergebnis *genau die Bits*, die *im Gray-Code* **`1`** sind
      (welche grundlegende Bitoperation leistet genau diese Bit-Invertierung?)
    - *Schiebe* den Gray-Code *um ein Bit nach rechts*.
- Weiters werden wir in den Punkten b) bis e) jeweils eine **Funktion** schreiben,
  die *die Gray-Codes berechnet* und *damit ein Array befüllt*. (auf *4 verschiedene Arten*).
  Diese Funktion hat *keinen Returnwert* und *zwei Parameter*:
  Das zu befüllende **`uint`**-*Array* und die *Anzahl der Bits* (*nicht* der Array-Elemente!).

Das **Hauptprogramm** soll wie folgt arbeiten:

- *Prüfe*, ob ein *Wert auf der Befehlszeile* angegeben wurde, und ob dieser *positiv* ist
  (sonst: Fehlermeldung, Programmende).
- Berechne aus dieser *Anzahl n der Bits* die *Anzahl* 2^n *der Codes*
  und leg ein lokales **`uint`**-*Array dieser Größe* an.

- Rufe die *Gray-Code-Berechnungsfunktion* auf, um dieses *Array zu befüllen*.
- Geh das Array mit einer *Schleife* durch und mache für jedes Element folgendes:
  - Gib in einer Zeile sowohl die *Element-Nummer* als auch den *Code aus dem Array* mit unserer Hilfsfunktion *als jeweils n-stellige Binärzahl* aus.
  - *Prüfe den Code* mit unserer *Rückrechnungs-Funktion* und gib eine *Meldung* aus, wenn er nicht stimmt.

b) Direkte Berechnung des Gray-Codes:

Die einfachste Möglichkeit, den Gray-Code zu berechnen, ist folgende:

- Nimm die Binärzahl **z**, deren Code du berechnen willst, und *halbiere* sie (*mit welcher Bitoperation* dividiert man eine Zahl durch 2 ?).
- Der zu berechnende Gray-Code hat *in genau jenen Bits* eine **1**, an denen die entsprechenden Bits in **z** und **z/2** *unterschiedlich* sind (es gibt eine Bitoperation, die genau das leistet. Verwende sie!).

Schreib zuerst eine kleine **Hilfsfunktion**, die genau das macht (mit **z** als *Parameter* und dem *Gray-Code* als *Returnwert*).

Schreib dann unsere **Array-Befüll-Funktion**, indem Du einfach in einer *Schleife* für jedes Array-Element diese *Hilfsfunktion aufrufst* (du musst zuerst wieder aus dem Anzahl-Bits-Parameter die Anzahl der Array-Elemente berechnen!).

c) Berechnung des Gray-Codes aus dem Gray-Code des Vorgängers:

Diesmal hat unsere **Hilfsfunktion** *zwei Parameter*: **z** (die *Binärzahl*, deren Gray-Code wir berechnen wollen) und **g** (den *Gray-Code* der *vorigen* Binärzahl).
Als *Returnwert* soll die Funktion wieder den *Gray-Code* von **z** liefern.
Die Funktion arbeitet wie folgt:

- *Suche* zuerst das *hinterste* **1**-Bit in **z**.
- *Invertiere* genau dieses eine Bit in **g**, um den neuen Gray-Code zu erhalten.

Die **Array-Befüll-Funktion** speichert zuerst einmal *im ersten Array-Element* den Wert **0**.
Für alle weiteren Elemente ruft sie in einer *Schleife* die *Hilfsfunktion* mit der *Element-Nummer* und dem *Wert des vorigen Array-Elementes* auf.

d) Reflexive Berechnung von Gray-Codes:

Diesmal haben wir *nur* die **Array-Befüll-Funktion**, *keine* Hilfsfunktion.

- Sie setzt zuerst das *erste Array-Element* auf **0**.
- Dann kommt eine *Schleife*, deren Zähler **b** *bei jedem Durchlauf verdoppelt* wird (**1**, **2**, **4**, **8**, **16** …), und zwar solange **b** *kleiner als die Anzahl* der Array-Elemente ist.

  Das Trickreiche ist, dass wir **b** *sowohl als Zahl als auch als Bitmaske* verwenden:

  - In einer *weiteren Schleife kopieren* wir jedesmal die *ersten* **b** *Elemente* des Arrays in die *zweiten* **b** *Elemente* des Arrays, und zwar *in umgekehrter Reihenfolge*, also **arr[0]** -> **arr[2*b-1]**, **arr[1]** -> **arr[2*b-2]** … **arr[b-1]** -> **arr[b]**.
  - Außerdem wird bei jeder Kopie *im Ziel-Element* zusätzlich *das Bit* **b** *gesetzt* (das Quell-Element der Kopie bleibt unverändert).

e) Rekursive Berechnung von Gray-Codes:

Die rekursive Berechnung von Gray-Codes verwendet dieselbe Idee wie Punkt c):
Der *nächste Gray-Code* wird *ausgehend vom vorigen Code* berechnet,
indem man *genau ein Bit invertiert*. Allerdings *entfällt die Suche mittels Schleife*
nach dem *"richtigen" Bit,* die in c) für jeden einzelnen Code wiederholt werden musste.

Stattdessen wird das zu ändernde Bit *durch die Rekursionstiefe* bestimmt:
Der *äußerste Aufruf* ist für die *Änderung des vordersten Bits* zuständig,
die *nächste Aufruf-Ebene* ändert das *zweitvorderste Bit,*
usw. bis zum *innersten Aufruf,* der jedesmal das *letzte Bit* ändert.

Anschaulich ergibt sich daraus folgender Ablauf:

- Schon *vor dem ersten rekursiven Aufruf* wird das *erste Element* initialisiert.
- *Jeder Aufruf* ist für 2^i*-1 Elemente* zuständig, denn das vorderste der 2^i Elemente bekommt schon seinen Wert, *bevor* der Aufruf der rekursiven Funktion erfolgt.
- Der Zerlegung "$2^i-1 = (2^{i-1}-1) + 1 + (2^{i-1}-1)$" folgend macht jeder Aufruf zuerst einen *rekursiven Aufruf* zum *Füllen der vordere Hälfte* seines Array-Bereiches, dann füllt er das *mittlere Element selbst,* und dann folgt der *zweite rekursive Aufruf* zum Füllen der *hinteren Hälfte* seines Array-Bereiches.

Die **rekursive Funktion** hat daher *drei Parameter*:
Das *Array,* eine *Bitmaske* **b** für das Bit, das *dieser Aufruf invertieren* soll,
und den *Index* **z** im Array, *ab dem* dieser Aufruf die Array-Elemente *füllen* soll.
Als *Returnwert* liefert die Funktion den *Index des nächsten zu befüllenden Elementes,*
damit der Aufrufer weiß, wo er im Array weitermachen muss.

- Ist **b** gleich **1** (*hinterstes* Bit = *innerster* Aufruf), so *füllt* die Funktion das Element **z** mit dem *Code aus dem Element* **z−1** mit Bit **b** *invertiert,* und *returniert* sofort **z+1**.
- Sonst kommt zuerst der *rekursive Aufruf* für die *vordere Hälfte* der Codes.
  Er erfolgt mit **b** *um eins nach rechts* geschoben und dem *ursprünglichen* **z**.
  Dieser Aufruf liefert als Returnwert die *neue Position* **z**.
- Dann *speichert* die Funktion in diesem Element **z** so wie oben
  den *Code aus dem Element* **z−1** mit Bit **b** *invertiert*.
- Zuletzt folgt der *rekursive Aufruf* für die *hintere Hälfte* ab *Position* **z+1**,
  wieder mit **b** *um eins nach rechts* geschoben.
  Die von diesem Aufruf gelieferte *neue Position* **z** wird *als Returnwert zurückgegeben*.

Die **Array-Befüll-Funktion** ist einfach: Zuerst das *erste Element* auf **0** setzen
und dann die *rekursive Funktion ab dem zweiten Element* aufrufen,
und zwar mit einem **b**, das dem *vordersten Bit der Codes* entspricht.

Zusatzaufgabe:

Dem Profi fällt auf, dass wir der rekursiven Funktion einen *Pointer auf den Array-Anfang* sowie einen *Index in das Array* übergeben, aber nirgends den numerischen Wert des Index und auch nirgends den Array-Anfang brauchen.

Bau die Funktion so um, dass der *Index-Parameter* **z** *wegfällt* und statt dem *Pointer auf den Array-Anfang* ein *Pointer auf das als nächstes zu befüllende Element* übergeben wird.
Auch der *Returnwert* ist dann kein Index, sondern der *Pointer auf das nächste Element*.