In het kort: teksten voor microcontrollers

spaar geheugen met compressie

Thomas Beck (Duitsland)

Veel microcontrollerprojecten hebben menu- en helpteksten en fout- of diagnosemeldingen die via een grafische gebruikersinterface of een diagnosepoort worden uitgevoerd. Veel tekst neemt uiteraard ook veel geheugenruimte in beslag. Als we echter geschikte compressie toepassen, dan kunnen we zoveel tekst opslaan in de controller zelf dat er geen extern geheugen meer nodig is. In dit artikel geven we een mogelijke oplossing: woordenboek-compressie.

Een voorwaarde voor elke succesvolle compressie is dat er enige redundantie zit in de data die gecomprimeerd moet worden. Redundantie in teksten ontstaat doordat bepaalde tekens vaker voorkomen dan andere, of doordat bepaalde tekenreeksen vaker voorkomen. Daardoor kunnen we gebruik maken van entropie-codering zoals Huffman-codering [4], waarbij letters die vaker voorkomen in een tekst worden gecodeerd in kortere bit-strings dan letters die minder vaak voorkomen. Een andere mogelijkheid is substitutie, zoals het LZ77-algoritme [5]. Dit heet ook wel woordenboek-compressie, en het komt erop neer dat herhaalde tekenreeksen in een woordenboek worden opgeslagen. In de tekst wordt zo'n tekenreeks dan vervangen door een verwijzing naar de plaats in het woordenboek (de 'ingang'). Het interessante van LZ77 is dat het daarbij niet eens nodig is om het woordenboek samen met de gecomprimeerde data op te slaan. In plaats daarvan vormt een gedeelte van de ongecomprimeerde data een woordenboek dat met voortschrijdende decompressie langzaam gevuld wordt en dat bij het bereiken van een maximale omvang middels een *sliding-window*-mechanisme wordt verschoven. Compressiesoftware zoals 7z en Zip gebruikt een combinatie van beide, namelijk woordenboek-compressie met daarna entropie-codering, voor nog sterkere compressie. Beide methoden leveren bij korte teksten echter weinig geheugenbesparing op. Bovendien kan men geen losse zinsnedes uit een gecomprimeerde tekst halen zonder de hele tekst te decomprimeren.

Analyse van de te comprimeren tekst

Listing 1 toont een gedeelte van een concrete applicatie. Een woordenboek-compressie die gebruik maakt van de redundantie van hele woorden en woordreeksen, belooft goede resultaten. De in elke tekst veel voorkomende spatie wordt als woordscheidingsteken gebruikt. De compressieroutine geeft dan de spatie een eigen woordenboek-ingang, of de decompressieroutine zet automatisch een spatie tussen elk woord.
Dit laatste verloopt echter alleen dan probleemloos als er geen bijzondere handelingen nodig zijn voor speciale tekens die zonder spatie voor of achter een woord staan – dus wanneer bijvoorbeeld *(Gauge)* één enkele ingang is en niet door de drie ingangen *(*, *Gauge* en *)* wordt gerepresenteerd.

- De hele tekst is beschikbaar in één of meer C-string-arrays ➞ losse teksten zijn gericht via array-indices worden geadresseerd, wat bij de gecomprimeerde tekst dus ook mogelijk moet zijn.
- De tekst bevat veel woorden en woordvolgordes die vaak voorkomen ➞ redundantie is een vereiste voor elke compressie.
- De tekst is in het Engels ➞ er komen alleen maar 7-bits ASCII-codes (0 … 127) in voor. Codes van 128 en hoger blijven vrij. Dat maakt stap 5 van de hier voorgestelde methode wat eenvoudiger.
- Veel woorden in de tekst beginnen met een hoofdletter ➞ Als dat niet zo was, dan zouden *Engine* en *engine* twee verschillende ingangen in het woordenboek geven. We krijgen dus verbeterde compressie als we alle zelfstandig naamwoorden met een hoofdletter laten beginnen. Omwille van de leesbaarheid worden voorzetsels en lidwoorden met een kleine letter geschreven.
- De tekst bevat weinig speciale tekens zoals *( ) # / - ,* ➞ er zijn dus weinig ingangen zoals *(Gauge)*, die afwijken van een mogelijk tweede ingang voor *Gauge*.
- De langste voorkomende tekst inclusief einde-string-teken is 57 tekens lang ➞ voor de decompressie is een buffer van minimaal 57 bytes RAM nodig.

Listing 1. Gedeelte van de te comprimeren tekst als C-string-array.

```
const char * const text[] = {
  /*   0 */ "Fuel System 1 Status",
  /*   1 */ "Fuel System 2 Status",
  /*   2 */ "Calculated Load Value",
  /*   3 */ "Engine Coolant Temperature",
  /*   4 */ "Short Term Fuel Trim Bank 1",
  /*   5 */ "Short Term Fuel Trim Bank 3",
  /*   6 */ "Long Term Fuel Trim Bank 1",
  /*   7 */ "Long Term Fuel Trim Bank 3",
  /*   8 */ "Short Term Fuel Trim Bank 2",
  /*   9 */ "Short Term Fuel Trim Bank 4",
  /*  10 */ "Long Term Fuel Trim Bank 2",
  /*  11 */ "Long Term Fuel Trim Bank 4",
  /*  12 */ "Fuel Pressure (Gauge)",
  ...
  /*  46 */ "OBD Requirements to Which Vehicle or
   Engine is Certified", // len=57
  ...
  /* 149 */ "Commanded Boost Pressure A",
  /* 150 */ "Boost Pressure Sensor A",
  /* 151 */ "Commanded Boost Pressure B",
  /* 152 */ "Boost Pressure Sensor B",
  /* 153 */ "Boost Pressure A Control Status",
  /* 154 */ "Boost Pressure B Control Status",
  ...
  /* 177 */ "Manifold Surface Temperature",
  /* 178 */ "Intake Manifold Absolute Pressure A",
  /* 179 */ "Intake Manifold Absolute Pressure B",
  ...
  /* 188 */ "Exhaust Gas Temperature Bank 2 -
   Sensor 7",
  /* 189 */ "Exhaust Gas Temperature Bank 2 -
   Sensor 8"
};
```

Hoe het allemaal begon...

De aanzet voor de ontwikkeling van verliesvrije tekstcompressie was mijn OBD2-voertuigdiagnoseproject OBD2 voor in Elektor [1][2][3]. Net als bij professionele diagnosetesters moesten er enige duizenden beschrijvingen komen voor diagnose-foutcodes (Diagnostic Trouble Codes, DTC's). De totale omvang van die teksten was ongeveer 211 KB. Met de hier gepresenteerde compressiemethodiek kon dat worden teruggebracht tot ongeveer 35 KB. Daardoor pasten de teksten in de AT90CAN128, de 8-bits AVR-microcontroller met 128 MB flash die ik voor dit project [1] gebruikt heb.
Voor de Arduino-versie [3] moest ook de kleinere Arduino Uno R3 en de Elektor Uno R4 met slechts 32 KB flash ondersteund worden. Deze geheugenruimte is natuurlijk ook voor de gecomprimeerde foutteksten te klein. De circa 7 KB aan beschrijvingen voor de tot dan toe ondersteunde OBD2 parameter identifiers (PID's) voor merendeels sensordata, kon ik echter comprimeren tot zo'n 2 KB. Als voorbeeld voor mijn compressie-algoritmedienen de 190 korte PID-teksten waarvan u in listing 1 een deel ziet.
Voor een 8-bits AVR-microcontroller AT90CAN128 en de avr-gcc-compiler 4.9.2 (optimalisatie -Os ingeschakeld) zijn de volgende waardes bepaald voor het geheugengebruik door de decompressieroutine (de andere waardes zijn berekend):

	Originele tekst [bytes]	Gecomprimeerde tekst [bytes]	Decompressie-routine [bytes]	RAM-buffer voor decompressie [bytes]
DTC-teksten	216311	35340	566	106
PID-teksten	7273	1820	314	57

Woordenboek-compressie

Stap 1: woordenboek aanleggen

In deze stap splitsen we de te comprimeren tekst op in losse woorden. Het scheidingsteken is een spatie, die later bij de decompressie weer wordt toegevoegd. Vervolgens worden de frequentie en de benodigde geheugenruimte (stringlengte plus einde-string-teken) van elk woord berekend. Daarna wordt het woordenboek gesorteerd in volgorde van aflopende woordfrequentie. Bij gelijke woordfrequenties komt het langste woord eerst. In **listing 2** ziet u alvast het woordenboek dat stap 2 oplevert, met frequentie en lengte in het commentaar.

Listing 2. Gedeelte van het geoptimaliseerde woordenboek als C-string-array.

```
const char * const dictionary[] = {
   /*   0 */ "Sensor", // cnt=116 len=7
   /*   1 */ "Bank", // cnt=104 len=5
   /*   2 */ "-", // cnt=89 len=2
   /*   3 */ "2", // cnt=67 len=2
   /*   4 */ "1", // cnt=66 len=2
   /*   5 */ "Fuel", // cnt=45 len=5
   /*   6 */ "Temperature", // cnt=43 len=12
   ...
   /*  16 */ "Exhaust Gas", // cnt=16 len=12
   ...
   /*  30 */ "Status", // cnt=9 len=7
   ...
   /*  45 */ "System", // cnt=4 len=7
   ...
   /*  77 */ "8", // cnt=2 len=2
   /*  78 */ "Currently Being Utilized by the",
   // cnt=1 len=32
   /*  79 */ "Advance for #1 Cylinder",
   // cnt=1 len=24
   ...
   /* 125 */ "F", // cnt=1 len=2
   /* 126 */ "G" // cnt=1 len=2
};
```

Listing 3. Array voor indexparen (berekend voor 8-bit indices).

```
const dict_index_t pair[][2] = {
   /*  0 => 127 */ {   2,   0 },
     // "-", "Sensor", cnt=78
   /*  1 => 128 */ {   1,   4 },
     // "Bank", "1", cnt=40
   /*  2 => 129 */ {   1,   3 },
     // "Bank", "2", cnt=40
   /*  3 => 130 */ { 128, 127 },
     // "Bank 1", "- Sensor", cnt=31
   ...
   /* 57 => 184 */ {  15,  51 },
     // "Air", "Flow", cnt=4
   ...
   /* 67 => 194 */ { 184,   0 },
     // "Air Flow", "Sensor", cnt=3
   /* 68 => 195 */ {  56, 194 }
     // "Mass", "Air Flow Sensor", cnt=3
};
```

Stap 2: woordenboek optimaliseren

Als er in de tekst woorden voorkomen met dezelfde frequentie die altijd naast elkaar staan, zoals *Exhaust* en *Gas*, dan resulteert dat in een enkele ingang: *Exhaust Gas*. Dat kost weliswaar een extra spatie tussen Exhaust en Gas, maar scheelt ook een einde-string-teken.

Na het samenvoegen van twee ingangen tot één, kunnen die nieuwe samengestelde ingangen opnieuw altijd dezelfde buurwoorden hebben. Als we de stap herhalen kunnen er daardoor ingangen van meer woorden ontstaan.

Aangezien er voor elke woordenboek-ingang (elke C-string-array) naast de geheugenruimte voor de string zelf ook geheugen voor een pointer naar die string nodig is, besparen we per samengestelde string een pointer. De daadwerkelijke besparing hangt af van de hoeveelheid geheugenruimte die een pointer kost op ons beoogde doelsysteem. Verder kunnen we zien dat de array `dictIndex[]` (die we in stap 3 genereren) hierdoor kleiner wordt. De afname is om precies te zijn: frequentie van de samengestelde ingang * `sizeof(dict_index_t)`.

Stap 3: codering van de oorspronkelijke tekst

Als het woordenboek klaar is, kunnen we de tekst coderen met behulp van de array-indices van de woordenboek-ingangen. De eerste tekst uit listing 1 ziet dat er dan zo uit:

text[0] = *Fuel System 1 Status*

- Fuel = ingang dictionary[5], vervang *Fuel* door index 5.
- *System* = ingang dictionary[45], vervang *System* door index 45.
- *1* = ingang dictionary[4], vervang 1 door index 4 (geen compressie).
- *Status* = ingang dictionary[30], vervang *Status* door index 30.

De spaties vallen weg. Dat levert de volgende array op:

```
const dict_index_t dictIndex[] = {
   /*   0 */ 5, 45, 4, 30, /* Fuel System 1 Status */
   /*   1 */ 5, 45, 3, 30, /* Fuel System 2 Status */
   …
   /* 189 */ 16, 6, 1, 3, 2, 0, 77 /* Exhaust Gas
   Temperature Bank 2 - Sensor 8 */
};
```

Aangezien het aantal indices per gecomprimeerde tekst variabel is, hebben we een tweede tabel nodig die voor elke tekst uit listing 1 een verwijzing naar zijn eerste index bevat. Zo kunnen we elke tekst onafhankelijk van voorgaande teksten decomprimeren.

```
const start_index_t startIndex[] = {
   /*   0 */    0, /* first index of text[0] is
   dictIndex[0] */
   /*   1 */    4, /* first index of text[1] is
   dictIndex[4] */
...
```

```
    /* 188 */ 1207, /* first index of text[189] is
    dictIndex[1207] */
    /* 189 */ 1213  /* first index of non-existent
    text[190] is dictIndex[1213] */
};
```

Het laatste element in array `startIndex[]` zorgt ervoor dat de decompressieroutine voor alle teksten, inclusief de laatste, het totale aantal indexen eenvoudig kan berekenen. Het aantal indexen voor text[n] is `startIndex[n+1] - startIndex[n]`.

Stap 4: het woordenboek uitbreiden voor indexparen

Deze stap comprimeert vaker optredende woordparen (indexparen) en woordvolgordes (indexvolgordes) nog wat effectiever. Als eerste zoeken we de meest voorkomende indexvolgordes die bestaan uit twee indices. Deze worden als nieuwe logische ingang toegevoegd aan het eind van het tot nu toe verkregen woordenboek. In **listing 3** ziet u de implementatie: we leggen een nieuw tweedimensionaal array `pair[][2]` aan en vullen dat met ingangen bestaande uit twee indices. Als de hoogste index in het woordenboek 126 was, dan krijgt de eerste index in `pair[0]` de logische index 127. Dan moeten we in array `dictIndex[]` alle indexparen die bestaan uit `pair[0][0]` en `pair[0][1]` (in listing 3 zijn dit index 2 en index 0) vervangen door een enkele index 127. De array `dictIndex[]` krimpt daardoor, de omvang van de afname is de grootte van dat indexpaar maal de frequentie, terwijl er in de array `pair[][2]` maar één ingang voor dit indexpaar bij komt. Deze stap laten we net zo vaak los op array `dictIndex[]` als er paren zijn die tenminste één keer voorkomen. Daardoor vinden we niet alleen twee woorden die naast elkaar staan, maar ook langere woordvolgordes. Een indexpaar kan dus zelf ook weer indices van een of twee indexparen bevatten (zie ingang 3, ingang 67 of ingang 68 in listing 3).

De frequentie van een paar moet voldoen aan de volgende voorwaarden om daadwerkelijk enige compressie op te leveren, c.q. om een nieuwe ingang in array `pair[][2]` op te leveren: frequentie indexpaar > 2 * `sizeof(dict_index_t)`.

Stap 5: ASCII-indices invoeren

In de daadwerkelijke implementatie volgt nog een verdere stap, die we hier vanwege zijn vele uitzonderingen en voorwaarden niet volledig kunnen behandelen. Door het invoeren van ASCII-indices voor woorden in het woordenboek die maar één keer voorkomen, of voor korte woorden met geringe frequentie die niet worden gebruikt in `pair[][2]`, kunnen we woorden die niet bijdragen aan de compressie uit het woordenboek verwijderen. Deze woorden zetten we direct ASCII-gecodeerd in array `dictIndex[]`, en we verwijderen ze uit het woordenboek. Van tevoren moeten we dan wel alle andere indices een offset hebben gegeven. Indices van louter 7-bits ASCII-codes verschuiven we bijvoorbeeld naar boven de 128. Meer gedetailleerde informatie hierover vindt u in de open source-code van de compressie- en de decompressieroutine bij dit Arduino-project [3].

Tot besluit

Bij de uitvoering van de afzonderlijke stappen kwam een algemeen compressieprobleem boven water. Voordat we gaan comprimeren weten we niet wat de benodigde geheugenruimte voor een enkele index `sizeof(dict_index_t)` is. Die waarde hebben we echter wel nodig om te kunnen uitmaken of een woordenboek-ingang voor korte of weinig voorkomende woorden überhaupt enige ruimtewinst oplevert. In het voorgestelde geval van tekstcompressie van PID-teksten zijn 8-bits indices voldoende, wat de methodiek en de decompressie wat eenvoudiger maakt. Maar tekstcompressie van de fout-teksten met maar 256 8-bits indices lukt niet. Daar passen we dan entropie-codering toe, die de meest frequente woorden codeert met 8-bits indices en alle andere woorden met 16-bits indices, om zo de groei van het `dictIndex[]`-array te beperken.

Als de tekstcompressie al na stap 3 of stap 4 volstaat, kunnen we het algoritme ook voortijdig beëindigen. De decompressieroutine neemt dan minder geheugenruimte in en kost natuurlijk ook minder verwerkingstijd dan wanneer we verdere stappen zouden zetten.

(180278-03)

Weblinks

[1] OBD2-Analyser NG: www.elektormagazine.com/labs/firmware-update-and-emulator-for-obd2-analyser-ng-wireless-obd2

[2] OBD2 for Raspberry Pi: www.elektormagazine.com/labs/obd2-for-raspberry-pi

[3] OBD2 for Arduino: www.elektormagazine.com/labs/obd2-for-arduino

[4] Huffman-codering: https://nl.wikipedia.org/wiki/Huffmancodering

[5] LZ77: https://de.wikipedia.org/wiki/LZ77