Turbo Pascal deel 6

MSX Club Magazine 36

Erik van Bilsen

Scanned, ocr'ed and converted to PDF by HansO, 2001

Een diverse inhoud deze keer: aangepaste procedures voor GIOS, machinetaal, geheugenmanipulatie, BASIC-equivalenten en recursief programmeren.

Procedurebibliotheken

In de afgelopen afleveringen hebben de diskabonnees verschillende procedurebibliotheken voor Turbo Pascal ontvangen. Deze bibliotheken hebben de vorm van include-files, en zijn genaamd GRAPH1 .LIB, GRAPH2.LIB en SOUNDS.LIB. Enige maanden geleden testte ik het Gra-phical Input Output System (GIOS) van de MSX club uit Enschede, waarvan u de bevindingen in dit nummer kunt lezen. Dit programma vervangt een groot deel van de procedures uit GRAPH1 .LIB, en heeft als bijkomend voordeel dat het weinig geheugen kost. Voor enthousiaste GlOS-gebrui-kers heb ik daarom de procedures uit de bestanden GRAPH1.LIB en GRAPH2.LIB die niet worden ondersteund door GIOS samengevat in het bestand GRAPHICS.LIB en aangepast voor GIOS. Het gaat dan voornamelijk om de smooth-scroll procedures. Over de methode die is gebruikt om de GlOS-procedures in een ander deel van het geheugen te zetten, kom ik wellicht in een volgende aflevering terug. Om met de GlOS-procedures en de procedures uit GRAPHICS.LIB te kunnen werken hoef je alleen GIOS te installeren en de volgende twee regels op te nemen in je programma:

```
{$1   MSXBIOS.LIB}
($1   GRAPHICS.LIB}
```

Het bestand GIOS.INC heb je niet nodig. De volgende procedures uit afleveringen 2 en 3 staan in GRAPHICS.LIB:

Screen	LogOpr
Setpage	PatternPaint
Cls	Text
Color	TextXY
GetPalet	SetAdjust
NewPalet	RamToVram
RestorePalet	VramToRam
Box	HHVram
Draw	InitVerScroll
Circle	InitHorScroll Ellipse

Natuurlijk kun je uit dit bestand die procedures verwijderen die je niet voor je eigen programma gebruikt.

Machinetaalprocedures

Machinetaal binnen Turbo Pascal: wat is hiervan het nut? Er zijn een paar redenen te noemen. Ten eerste heeft TP een aantal beperkingen. Zo zijn er geen standaardprocedures om direct het geheugen en I/O-poorten te lezen en te schrijven. Een andere reden is snelheid. Turbo Pascal is al erg snel omdat het een compiler is, maar machinetaal blijft altijd sneller. De moet hier trouwens wel even een opmerking maken over het willekeurig gebruik van de termen 'machinetaal' en 'assemblertaal'. Deze twee talen zijn niet hetzelfde! Het trieste is dat vaak het woord machinetaal wordt gebruikt, als er eigenlijk assemblertaal wordt bedoeld.
Wat is dan het verschil tussen machinetaal en assemblertaal ? In het kort: machinetaal is 0 0 11 1110 0110 0100 (eventueel 3E 64) en assemblertaal is ld a, 100. Machinetaal stamt uit de tijd dat computers werden geprogrammeerd door het aan- (1) en uitzetten (0) van schakelaars, en behoort sinds de komst van het toetsenbord en assemblers eigenlijk tot de verleden tijd. De twee talen hebben wel veel gemeen: ze staan beide het dichtst bij de machine. Assemblertaal wordt immers direct vertaald naar machinetaal; elke assembler-instructie heeft een equivalent in nulletjes en eentjes. Bij de andere talen is dit niet het geval. Denk dus twee keer na voordat je het woord machinetaal gebruikt!

De InLine-procedure

Genoeg uitgeweken. Machinetaalprocedures kunnen in TP-tekst worden opgenomen met de standaard InLine-procedure. Met deze procedure kun je rechtstreeks getallen in de programmacode voegen. De regel InLine (1/2/3) zet bijvoorbeeld de getallen l, 2 en 3 in het gecompileerde programma. Een voorbeeld is de Sound-proce-dure uit nummer 37, die onderaan de vorige pagina nog eens is herhaald. Met behulp van deze procedure kan naar een PSG-register worden geschreven met behulp van de daarvoor bestemde I/O poorten. Zoals je ziet is elk getal hexadecimaal weergegeven en worden ze gescheiden door een schuine streep (/). Tevens valt op dat in de InLine-procedure twee namen van variabelen zijn verwerkt. Op deze plaatsen wordt het adres van de variabelen geplaatst. Staat bijvoorbeeld de variabele Register op adres 50000, dan wordt het getal 50000 ingevoegd. Op dezelfde manier kunnen ook procedurenamen in de InLine-procedure worden verwerkt. Op deze plaats wordt dan het aanroepadres van de procedure geplaatst. Als we de Sound-procedure terugvertalen naar assemblertaal, krijgen we het volgende:

```
ld  c,160
ld a,(Register)
out   (c),a
inc  c
ld a,(Data)
out   (c),a
ei

PROCEDURE  Poke   (Adres:   INTEGER;   Data:   BYTE);
```

```
BEGIN
  InLine ($3a/Data/  { ld a,(Data) }
          $2a/Adres/ { ld hl,(Adres) }
          $77/       { ld (hl),a }
          $fb)       { ei }
END ;

FUNCTION Peek (Adres: INTEGER):BYTE;

VAR Dummy: BYTE;

BEGIN
  InLine ($2a/Adres/    { ld hl,(Adres) }
          $7e/          { ld a,(hl)     }
          $32/Dummy/    { ld (Dummy), a }
          $fb);         { ei           }
  Peek:=Dummy;
END;

PROCEDURE iPoke (Adres, Data: INTEGER);

BEGIN
  Poke (Adres,Lo(Data));
  Poke (Adres+1,Hi(Data))
END;

FUNCTION iPeek (Adres: INTEGER):INTEGER;

BEGIN
  iPeek:=Peek(Adres)+Peek(Adres+l) shl 8
END;

PROCEDURE rPoke (Adres: INTEGER; Data: REAL);

VAR Dummy: INTEGER;

BEGIN
  Dummy:=Addr(Data);
  iPoke(Adres,  iPeek(Dummy  )};
  iPoke(Adres+2,iPeek(Dummy+2));
  iPoke(Adres+4,iPeek(Dummy+4))
END;

FUNCTION rPeek (Adres: INTEGER):REAL;

VAR Dummy: REAL;
    DumAdr: INTEGER;

BEGIN
  DumAdr:=Addr(Dummy);
  iPoke(DumAdr  ,iPeek(Adres  ));
  iPoke(DumAdr+2,iPeek(Adres+2));
  iPoke(DumAdr+4,iPeek(Adres+4));
  rPeek:=Dummy
END;
```

De procedure wordt afgesloten met de instructie 'ei' (enable inter-rupt, hexadecimaal $FB). Zoals je ziet wordt de procedure niet afgesloten met een ret-instructie ($C9). Dit hoeft ook niet, want het woord END aan het einde van de Sound-procedure wordt bij het compileren al vertaald in een ret-instructie.

Peek en Poke

Met behulp van de InLine-proce-dure kunnen op deze manier ook equivalenten van de PEEK- en POKE-instructies worden gemaakt. Zie het voorbeeld bovenaan de vorige pagina. De vertaling naar as-semblertaal is als commentaar toegevoegd. Nu we toch een poke-procedure en een peek-func-tie hebben, kunnen we deze procedures net zo goed uitbreiden met procedures voor het peeken en poken van INTEGERS. De oplossing is te vinden in de procedures iPoke en iPeek. De procedure iPoke maakt weer gebruik van Poke om het lage (Lo) en hoge (Hi) deel van een integer in het geheugen te zetten. Gaan we nog een stapje verder, dan maken we dezelfde procedures voor reële getallen. Het resultaat is de procedures rPoke en rPeek. In TP is een REAL 6 bytes (ofwel 3 integers) groot. Een reëel getal kan dus worden gepoked door drie maal de procedure iPoke aan te roepen. Maar hoe zetten we een reëel getal om in 3 integers? Hiervoor passen we een truuk toe met de standaardfunctie Addr. Deze procedure toont veel gelijkenissen met de BASIC-instructie VARPTR. De functie Addr wordt aangeroepen met de naam van een variabele of procedure, en geeft het adres van die variabele of procedure terug. In de procedure rPoke staat de volgende regel:

```
Dummy:=Addr(Data);
```

Na deze regel bevat de variabele Dummy het adres van de reële variabele Data. Bevat Dummy bijvoorbeeld de waarde 10000, dan is de variabele Data opgeslagen op de geheugenadressen 10000 tot en met 10005. Probeer het volgende programmaatje maar eens:

```
rPoke (50000,1.234) ; WriteLn (rPeek(50000))
```

Read en Data

Een nadeel van Turbo Pascal is, zoals enkele lezers vast al hebben gemerkt, dat er geen variant is van de BASIC-instructies READ en DATA. Diezelfde lezers zien met de InLine-procedure misschien al een oplossing. Als we deze procedure kunnen gebruiken om rechtstreeks machinecode in het programma te zetten, dan moet die immers ook te gebruiken zijn om dataregels in het programma te zetten. De oplossing is als volgt zet de dataregels met behulp van InLine in een procedure met bijvoorbeeld de naam Data. Deze procedure moet je dus nooit aanroepen! Met behulp van de Peek-functie kunnen deze gegevens dan uitgelezen worden. Het voorbeeld bovenaan deze pagina bevat een BASIC-programma met bovengenoemde oplossing in Turbo Pascal. Op deze manier kunnen ook ar-rays worden gevuld. Als een array maar één keer gevuld hoeft te worden, kunnen we gebruik maken van een snellere methode. Hierbij wordt een array als een constante gedefinieerd op de volgende manier:

```
CONST
 A:   ARRAY    [0..4]    OF BYTE=(12,34,56,78,90)
```

Recursief programmeren

In nummer 37 heb ik al even het begrip recursief programmeren aangestipt. Ik heb het daarbij omschreven als een verschijnsel van een procedure of functie die zichzelf aanroept. Ook in BASIC kun je recursief programmeren, alleen is daarover weinig bekend. De meest eenvoudige vorm van een recursief programma in BASIC is het volgende:

```
10   N   =   N   +   l
20   GOSUB   10
```

Na het starten van dit programma verschijnt op den duur de foutmelding 'Out of memory in 107. Dit gebeurt op mijn MSX als N de waarde 3318 heeft bereikt. Hoe komt dit nou? ledere keer als met behulp van GOSUB een subroutine wordt aangeroepen, wordt de huidige waarde van de programmateller (het adres van de huidige instructie) onthouden. Deze waarde wordt op de zogenaamde 'stapel' (stack) gezet. Als de BASIC-inter-preter de instructie RETURN vindt, wordt die waarde weer van de stapel gehaald en in de programmateller gezet. In het bovenstaande programma staat echter geen RETURN, met als gevolg dat stapel als maar groeit totdat het geheugen vol is, in dit geval dus na 3318 keer.

Sigma en faculteit

Standaardvoorbeelden van recursieve functies zijn de sigma- en de faculteit-functie. De sigma-functie (E) berekent de som van gehele getallen van l tot n. IA is dus 1+2+3+4=10. Evenzo is £5=15 want £5=Z4+5. De recursieve Sigma-functie is te vinden bovenaan de pagina.
Zoals je ziet wordt na ELSE opnieuw de Sigma-functie aangeroepen met de parameter Getal-1. Dit gebeurt net zolang totdat Getal kleiner of gelijk is dan 1. We volgen de functie stap voor stap:
De functie wordt aangeroepen met WriteLn (Sigma(4)). Het getal is groter dan l, dus wordt opnieuw de functie sigma aangeroepen: Sigma := 4 + Sigma(3), totdat de waarde l wordt bereikt:

1. Sigma := 4 + Sigma (3)
2. 2. Sigma := 3 + Sigma (2)
3. 3. Sigma := 2 + Sigma (1)
4. 4. Sigma := l

Nu wordt teruggerekend om het antwoord te krijgen:

Sigma(l) := l

dus Sigma(2) := 2 + Sigma(l) wordtSigma(2) := 2 + l (=3) Hieruit volgt dat Sigma(3) := 3 + Sigma(2) wordt Sigma(3) := 3 + 3 (=6) en Sigma(4) := 4 + Sigma(3) dus Sigma(4) := 4 + 6 (=10)

Recursie of iteratie?

Natuurlijk kan de Sigma-functie ook worden geprogrammeerd zonder gebruik te maken van recursie. Een voorbeeld daarvan staat onderaan de pagina. Waarom dan toch gebruik maken van recursie? Ten eerste is recursief programmeren vaak korter. Ten tweede zul je in gestructureerde talen als (Turbo) Pascal en C vaak recursieve procedures tegenkomen omdat recursief programmeren een zeer gestructureerde manier van pro-bleemoplossen is. Recursie wordt bijvoorbeeld gebruikt voor problemen die bestaan uit soortgelijke deelproblemen, zoals de Sigmafunctie. Het 'probleem' Sigma(5) bestaat uit een deelprobleem van soortgelijke aard, namelijk 5+Sig-ma(4).
Recursie, geen gemakkelijk onderwerp om te begrijpen en om over te schrijven. Maar wil je volledig zijn, dan ontkom je er niet aan het onderwerp aan te snijden. De volgende aflevering is weer wat meer gericht op praktische toepassingen. Je krijgt dan in ieder geval een overzicht van de behandelde standaard procedures en -functies en, als ik niet te krap in de tijd zit, een grafisch kadootje: screen 9, een softwarematig scherm met een resolutie van 1024 * 1024 speciaal voor de printer!