

# Programmeren in de Wimp

Een cursus programmeren in drie delen voor de RISC OS-omgeving

```
CASE deel% OF
  WHEN 1: PROCditboek
  WHEN 2: PROCvolgendboek
  WHEN 3: PROClaatsteboek
ENDCASE
```

Peter Scheele



## Voorwoord

Dit boek is bedoeld voor degenen die willen leren hoe het programmeren in de Wimp in z'n werk gaat. Wimp staat voor Windows, Icons, Menus and Pointers die de basisingrediënten vormen van de grafische gebruikers interface (GUI) van RISC OS.

Het boek bestaat uit drie delen: 'Leerling', 'Gezel' en 'Meester'.

Deel 1 'Leerling' is bedoeld om inzicht te krijgen in de principes van multitasking onder RISC OS. Verder behandelt het de opzet van een programma; er wordt een werkend geheel opgezet dat kan dienen als skelet want het moet verder worden uitgebreid met functionaliteiten die zorgen dat het ook een bruikbaar programma wordt. Tot slot van dit deel is er aandacht voor aspecten van software ergonomie.

In deel 2, 'Gezel', worden op thematische wijze functies aan het skelet toegevoegd. Niet vanwege de functionaliteit, maar om de mogelijkheden van de Wimp beter en uitgebreider te kunnen benutten. Het uitgangspunt wordt steeds gevormd door cases waarin een situatie wordt beschreven die programmeervragen oproept. De opbouw van de cases is steeds een situatiebeschrijving waarin een programmeerprobleem aan de orde is, vervolgens wordt de vraag gesteld die in de case beantwoord moet gaan worden, dan wordt in denkstappen uiteengezet hoe in grote lijnen de oplossing gezocht moet worden, er wordt een lijst van benodigdheden gemaakt en tot slot vindt de uitwerking plaats.

In deel 2 komen diverse Wimp-functies aan de orde.

In deel 3, 'Meester', gaat het om grotere projecten. Hoe is de ontwikkeling van probleem tot programma. Hoe kan een ietwat vaag idee worden ontwikkeld tot een bruikbaar programma. Op welke manier is het mogelijk toegang te krijgen tot gegevensverzamelingen en kan er een gebruikersinterface voor worden geschreven. Er zullen voorbeelden gegeven worden, maar het is ook de bedoeling te rade te gaan bij enkele ervaren programmeurs om te zien hoe zij het doen.

Dit boek is tot stand gekomen via de mailinglist Acornisten[Program]. Ik wil vooral Ed van der Meulen bedanken voor zijn kritische opmerkingen op de momenten dat ik te onnauwkeurig was in mijn beschrijvingen.

Deze lesteksten zijn public domain. Dat wil zeggen dat ze voor ieder gratis ter beschikking zijn, behoudens de kosten van het medium. Er rust echter wel auteursrecht op, ongeacht het medium.

Maastricht, 2000-2001

Bij de herziene uitgave van 2006

Ik heb de lesteksten beter gestructureerd, met wat meer voorbeelden. Het is daardoor, zo hoop ik, gemakkelijker geworden om te lezen en toe te passen.

Peter Scheele ©

Maastricht, 2006

p.scheele@hccnet.nl

1	Hulpmiddelen bij het programmeren onder RISC OS	<b>1</b>
2	Multitasking onder RISC OS	<b>3</b>
	2.1 Hoe multitaskt RISC OS	3
3	De programmadirectory	<b>5</b>
	3.1 de programmadirectory	5
	3.2 de !Boot-file	5
	3.3 de !Run-file	7
	3.4 een sprite-file	7
	3.5 de !Help-file	10
	3.6 de !RunImage	10
4	!RunImage-building; de opzet van het hoofdprogramma	<b>11</b>
	4.1 de SWI's	11
	4.2 parameters	13
	4.3 het hoofdprogramma	13
	4.4 de WIMP-poll	14
	4.5 het eigenlijke pollen	16
	4.6 de parameterblokken	18
	4.7 converteren van binair naar hexadecimaal	21
5	Templates	<b>23</b>
	5.1 een beeld van een venster	23
	5.2 de template-editor	24
	5.3 de vereniging van template en programma	24
	5.4 de praktijk	27
	5.5 een eenvoudige foutprocedure	28
6	Interactie met het programma: de muis met knoppen en klikken en toetsenbordinput	<b>31</b>
	6.1 de muis	31
	6.2 uitvoer in iconen	33
	6.3 invoer uit iconen	35
7	De icoonbalk en menu's	<b>39</b>
	7.1 de icoonbalk	39
	7.2 menu's	40
8	Het opsporen en afhandelen van fouten	<b>47</b>
	8.1 de syntactische fouten	47
	8.2 de run-time errors	48
	8.3 de logische fouten	49
	8.4 systematisch testen	49
	8.5 wat hoef je niet te testen?	50
	8.6 het afhandelen van fouten	50
	8.7 meldingen	51
	8.8 communicatieve fouten	51



Waaruit bestaat het gereedschap van de (beginnende) WIMP-programmeur?

Om te beginnen natuurlijk BBC-Basic of, zoals het nu heet Basic V. Maar dat is in ROM aanwezig en dat kunnen we zo activeren. We zullen in hoofdstuk 3 zien hoe dat moet.

BBC-Basic

We moeten tekstbestanden kunnen maken en daarvoor gebruiken we een editor. !Edit voldoet, maar !Zap of !StrongEd zijn wat handiger. Neem een editor waar je het plezierigst mee werkt. Ook !Writer doet het, al moet je je bestanden dan als Text bewaren en er vervolgens het bestandstype van maken dat nodig is.

!Edit  
!Zap  
!StrongEd  
!Writer

Voor het maken van sprites hebben we !Paint, hoewel ook !Draw van pas kan komen. Het is ook mogelijk geavanceerdere programmatuur te gebruiken als !ArtWorks of !Photodesk. Bij de vectorprogramma's moeten we de mogelijkheid hebben om van de vectorafbeelding een bitmap te maken. Daar zijn conversieprogramma's voor maar !Snapper en !Paint kunnen het ook, zij het op een andere manier: ze maken er een foto van. Het komt in hoofdstuk 3 aan de orde.

!Paint  
!ArtWorks  
!Photodesk  
!Snapper

Voor het maken van templates hebben we een template-editor nodig. !TemplEd, !FormEdExt, !FormEd, !WindowEd of !WinEd zijn mogelijk. !TemplEd is geleverd op de cd-rom voor RISC OS 4 en is dus freeware. Het is wel handig als we die voor de cursus gebruiken, want dan praten we over hetzelfde. De templates gaan we in hoofdstuk 6 behandelen.

!TemplEd



Het grote naslagwerk op dit gebied is de Programmers Reference Manual (PRM). Het hoeft niet eens de meest recente versie te zijn. Probeer er desnoods voor een tijdje een te lenen. Misschien kun je er een op de Acorn Expo (tweedehands) kopen. Ook !StrongHelp is bruikbaar. Met de onderdelen OS SWI, SWI of WIMP SWI's kom je ook een heel eind al is de PRM uitvoeriger en overzichtelijker. StrongHelp is te vinden via Google, evenals de StrongHelp-bestanden.

PRM

!StrongHelp

Het BBC-Basic handboek voor allerlei programmeervragen. Ook daarvoor zijn StrongHelpversies als Basic en VDU. En ook hiervoor geldt dat het boek (er is ook een engelstalige versie) wat handiger in het gebruik is.

BBC-handboek

Het boekje WIMP Programming van Lee Calcraft kan goede diensten bewijzen, maar is niet noodzakelijk. Dit boek is overigens uitverkocht. Eerste editie was 1992, de tweede in 1993.

WIMP  
Programming  
for All

De RISC OS Styleguide. Daarin wordt ondermeer geschreven over aspecten van software-ergonomie. Het zegt iets over de kleuren van vensters, balken en knoppen, afstanden tussen knoppen, eenduidigheid van berichten etc. Niet noodzakelijk, maar wel belangrijk als je je programmavensters goed wilt vormgeven.

RISC OS  
Styleguide

Wat kennis en vaardigheid in het programmeren. Je moet weten hoe je lussen maakt, procedures of functies aanroept met parameters, arrays en DIM's definieert, geheugenoperatoren gebruikt etcetera. Hoe dat alles in de WIMP wordt toegepast, komt in deze cursus aan de orde.

kennis en  
vaardigheden

- Utilities            Verder zijn er tal van utilities die het leven vergemakkelijken. Debuggers om fouten op te sporen, crunchers om overbodige spaties of commentaar te verwijderen, Windowflag-generators, ja zelfs programma's om (bijna) volledige Basicprogramma's mee te genereren. In eerste instantie zullen we echter het handwerk moeten leren.
- Aanpak             Natuurlijk is er een vorm van aanpak nodig. Je kunt altijd terugvallen op het experiment. Probeer eens wat, kijk hoe anderen het gedaan hebben en leer ervan. Vaak is het handig de zaken gestructureerd aan te kunnen pakken. Teken vooraf op papier het venster zoals je het hebben wilt en maak het later na met een template-editor. Schrijf eerst op papier dat moeilijke stukje programma en test het, in theorie, voordat je het invoert. We besteden daar aandacht aan in hoofdstuk 8.
- Idee                En, we vergeten het haast, een tamelijk gedetailleerd idee. Een mooi spelletje, een utilitie, een serieuze toepassing om het leven, of op zijn minst het computergebruik, te veraangemen. Dat hoeven geen grote ideeën te zijn. Begin klein om ervaring op te doen en de grotere projecten komen later vanzelf.
- Succes ermee.

In dit hoofdstuk staan de volgende paragrafen:

### 2.1 Hoe multitaskt RISC OS 3

Je kunt een programma maken dat de computer geheel overneemt. Het is dan niet mogelijk tegelijkertijd andere programma's te laten draaien. We noemen dat singletasking, omdat de processor en het besturingssysteem zich alleen met dat programma hoeven bezig te houden. Het programmeren op de BBC of vergelijkbare systemen deed zich zo voor.

singletasking

Wie echter zijn programma naast andere programma's wil laten werken, moet zich realiseren dat het programma zich dan dient te schikken naar de regels die het OS moet toepassen om multitasking snel en veilig te kunnen uitvoeren. Om die regels te begrijpen, is het nodig iets te weten van de wijze waarop RISC OS die multitasking uitvoert.

multitasking

### 2.1 Hoe multitaskt RISC OS

Stel je voor dat het OS voortdurend een lus doorloopt. In die lus zitten taken. Die taken kunnen programma's zijn, ook dat van jou. Iedere taak is op een zeker moment aangemeld. Sommige tijdens de startprocedure van het systeem, andere later.

Allemaal hebben ze bij hun aanmelding hun identiteit kenbaar gemaakt en kregen ze een uniek nummer. Zo werkt het OS die taken af en begint weer opnieuw, honderden, zo niet duizenden keren per seconde. Nu heeft iedere taak bij zijn aanmelding ook kenbaar gemaakt wat het van het OS verwacht. Iedere keer dat de taak in de lus aan de beurt is, vraagt de taak aandacht als het nodig is en geeft het OS specifieke informatie door aan die taak. Het OS verzorgt namelijk ook een belangrijk deel van in- en uitvoer. Als jij met de muis beweegt, een muisknop indrukt, het toetsenbord bedient of er op het scherm in een van de vensters iets gedaan wordt, signaleert het OS dat en geeft dat door aan dat programma waar dat betrekking op heeft. En juist daar zul je in jouw programma extra aandacht voor moeten hebben. RISC OS treedt namelijk op als tussenpersoon tussen de gebruiker en het programma en zorgt ervoor dat vanalles van de een wordt doorgegeven aan de ander of, omgekeerd, van de ander aan de een.

identiteit

Dit heeft een aantal consequenties

Omdat er meer programma's tegelijk werken, zijn ze ook tegelijk aanwezig in het interne geheugen. Daarbij moet voorkomen worden dat een programma een ander programma in de weg zit. Het mag niet voorkomen dat een programmadeel wordt overschreven door een deel van een ander programma. Daarvoor is een vorm van protectie nodig. De programmeur heeft daar voor een deel voor te zorgen. Die kan dat doen door, door middel van WimpSlot, minimum en maximum grenzen op te geven van het geheugen dat door het programma geclaimed moet worden. We komen er in hoofdstuk 3 op terug als we een !Run-file gaan maken.

Geheugen-  
protectie  
WimpSlot

Bij het opzetten van een programma moeten we aandacht besteden aan de aanmelding en identificatie van het programma. En als het programma beëindigd wordt, zullen we het moeten afmelden. Verder zullen we zorgvuldig moeten aangeven welke informatie wel en welke informatie niet doorgegeven zal worden door het OS. Als we alles toestaan, vertraagt dat de multitasking en daarmee de snelheid van ons programma. We komen daar in hoofdstuk 4 op terug.

Systeem-  
aanroepen

Om de multitasking snel en efficiënt te laten werken zijn er allerlei standaardvoorzieningen aanwezig voor invoer, uitvoer, schermafhandeling etc. Daarbij hoort bijvoorbeeld ook een uitgebreide definitie van vensters. Die definitie zit in ROM en door middel van systeemaanroepen kan die worden geactiveerd. We kennen daar uitgebreide aanroepen voor met een heleboel parameters, maar die gebruiken we bij voorkeur zo weinig mogelijk. In plaats daarvan nemen we de Template-editor en we zullen zien dat daarmee allerlei vensterattributen kunnen worden aangezet. Denk daarbij aan schuifbalken, titelbalk en de knoppen op vier hoeken van het venster. We hoeven die niet meer in het programma aan te geven, ze hebben een standaardwerking en de afhandeling ervan, als de gebruiker erop klikt, gebeurt vanzelf. Zo kunnen we ook invoervelden aangeven (eigenlijk zijn het iconen of nog beter writeable icons) en als er iets wordt ingevoerd, signaleert het OS dat. We hoeven dus nooit meer gebruik te maken van INPUT of een vergelijkbaar statement. Ook PRINT of PRINT TAB() is niet meer nodig. Maakte je vroeger een invoerlus als:

```
REPEAT
  INPUT a$
UNTIL a$="Klaar"
```

Hoofdlus

tegenwoordig handelt het OS dat goeddeels af en zijn zulke invoerlussen zelfs funest voor de multitasking. In hoofdstuk 4 zullen we zien dat er een hoofdlus gebouwd zal worden waarin veel invoer en uitvoer verwerkt wordt. Als je met zo'n invoerlusje zoals hierboven de werking van de hoofdlus verstoort, verstoort je ook de multitasking en daar hebben andere programma's last van.

Ergens las ik de zin dat jouw programma zo weinig mogelijk intelligente beslissingen moet nemen als maar mogelijk is. Dat geldt dan voor alles dat met RISC OS te maken heeft. Laat het werk opknappen door dat deel van het systeem dat daarvoor gemaakt is. Als je het anders doet, vertraagt of verstoort dat misschien de werking van het systeem.

berichten

drag-and-drop

Een ander aspect is dat via het OS berichten van de ene toepassing kunnen worden doorgegeven aan een andere toepassing. Daarbij moeten we het woord berichten ruim opvatten; het kunnen zelfs tekstfragmenten of bestanden zijn. We spreken over drag-and-drop, een bijzondere voorziening die onder RISC OS op bijzonder efficiënte wijze toegepast kan worden.

SoftWare Interrupt (SWI)

Omdat er zoveel taken en deeltaken door RISC OS zelf kunnen worden uitgevoerd, zijn programma's ook zo klein. We weten het, een toepassing die te groot is voor een diskette, is een zeldzaamheid. Het aantal taken, we noemen ze software interrupts of SWI's, is groot. Enkele honderden. Dat het allemaal in vier Mb ROM past, is verbazingwekkend. Maar wij programmeurs m/v zullen er de juiste uit moeten kiezen, om die op het juiste moment te laten uitvoeren. Daar zit een moeilijkheid, want eigenlijk zou je een duidelijk beeld moeten hebben van alle SWI's om precies te weten waar wat op welke manier moet gebeuren. In deze cursus zal een klein deel ervan aan de orde komen. Het is wel dat deel dat je in vrijwel ieder programma tegenkomt omdat ze daar de ruggengraat van vormen.

programma-  
directories

Wie vaker in programmadirectories gekeken heeft (met de Shift-toets ingedrukt klikken), heeft gezien dat er in zo'n directory verschillende bestanden aanwezig zijn. De meeste bestanden zijn ervoor om de voorzieningen voor het eigenlijke programma te treffen.

In het volgende hoofdstuk gaan we beginnen met het maken van zo'n programmadirectory en zullen we zien welke bestanden dat zijn en hoe ze gemaakt worden.

In dit hoofdstuk staan de volgende paragrafen:

- 3.1 de programmadirectory 5
- 3.2 de !Boot-file 5
- 3.3 de !Run-file 7
- 3.4 een sprite-file 7
- 3.5 de !Help-file 10
- 3.6 de !RunImage 10

Om een programma te maken, dat zich gedraagt als andere programma's, zullen we de omstandigheden waaronder dat programma zich gedraagt moeten vastleggen. We zullen daarvoor de volgende stappen moeten doorlopen:

- 3.1 maak een programmadirectory en open die
- 3.2 maak met je editor een obeyfile met de naam !Boot
- 3.3 maak met je editor een obeyfile met de naam !Run
- 3.4 maak een spritefile met de naam !Sprite
- 3.5 maak met je editor een textfile met de naam !Help
- 3.6 maak met je editor een Basicfile met de naam !RunImage

Hoe pakken we dat aan?

### 3.1 de programmadirectory

We gaan eerst een programmadirectory maken en we zullen daar de verschillende bestanden in gaan plaatsen. Zet de muispijl in een directoryvenster en druk op menu (de middelste knop). Ga bij 'New directory' naar rechts en voer een naam in die begint met een uitroepteken. Laten we het !MijnProg noemen. Druk dan op OK. De nieuwe directory is nu klaar en staat in de directory met het icoon App weergegeven. Open met Shift-klik die directory.



### 3.2 de !Boot-file

De !Boot is er om in een directoryvenster de iconen van de toepassingen zichtbaar te maken, als een toepassing een icoon heeft. RISC OS loopt in die directory alle !Boot-files vlak voor het openen na en kan dan zien waar het icoon zich bevindt. Het icoon is overigens meestal een sprite. De !Boot is, net als de !Run, een bestand van het type 'obey'. Er zijn twee manieren waarop je een obey-file kunt maken: direct en achteraf.



Eerst de directe manier.

Start je editor en wijs met je muispijl het programma-icoon ervan op de icoonbalk aan. Druk op menu en kies 'Create' uit het menu en ga daar naar rechts. Kies vervolgens voor Obey. Er verschijnt een leeg tekstscherf en daarin kan de tekst voor de !Boot-file komen te staan. Verderop zal ik aangeven wat die tekst zal zijn. Bewaar het bestand onder de naam !Boot in het daarstraks geopende venster van je programma.

De manier achteraf is dat je je editor activeert en een leeg tekstscherf opent. Ook nu voer je de tekst in en bewaart die onder de naam !Boot in je programmavenster. Maar het is nog een bestand van het type Text. Zet je muispijl op het icoon van !Boot en klik menu. In het menu zie je "File '!Boot'" met een pijltje naar rechts. Volg dat pijltje en er verschijnt een menu. Daarin staat onderaan 'Set type' met een pijltje naar rechts. Volg dat ook en je ziet een invoerveld waarin 'Text' staat. Maak het veld leeg met CTRL-U en voer er 'Obey' in. Afsluiten met een druk op de Return-toets en de zaak is klaar.

Merk op dat je aan het bestandsicoon kunt zien of het van het type Text of Obey is.

Wat komt er in de !Boot-file te staan?

Om te beginnen twee regels (al komen er later nog wat bij):

```
Set MijnProg$Dir <Obey$Dir>
IconSprites <MijnProg$Dir>!Sprites
```

De uitleg van de eerste regel is dat je hiermee aangeeft waar RISC OS bestanden kan vinden die in jouw programmadirectory staan. Overweeg de volgende twee mogelijkheden: de concrete en de abstracte.

De concrete padaanduiding beschrijft de weg van discfilingsystem, via drive tot bestand. Dat is bij mij

```
ADFS::IDEDisc5.$!MijnProg.!Sprites
```

Je begrijpt dat ik mijn programmadirectory niet moet verplaatsen naar een andere drive of een andere directory omdat dan het pad niet meer klopt en het bestand !Sprites niet meer te vinden is.

Daarom de abstracte. Ik geef een variabele aan, (MijnProg\$Dir) en RISC OS zet daar het huidige pad in. Voor RISC OS zijn dan MijnProg\$Dir en ADFS::IDEDisc5.\$!MijnProg dezelfde. Maar MijnProg\$Dir en SCSI::Disc7.\$!MijnProg is dan ook geldig als mijn programma daar zou staan. De naam van de variabele mag je zelf kiezen al is het wel verstandig te laten zien bij welk programma het hoort.

De tweede regel geeft aan dat het bestand !Sprites staat in het zojuist aangegeven pad. We doen dat door de variabele op te geven en daarachter .bestandsnaam te vermelden. Maar stel je voor dat je besloten hebt om allerlei bestanden in een aparte directory te stoppen met bijvoorbeeld de naam Bestanden. Dan wordt de regel:

```
IconSprites <MijnProg$Dir>.Bestanden.!Sprites
```

En als ik in Bestanden veel files heb die regelmatig gebruikt moeten worden, kan ik ook een tweede variabele -met een eigen variabelenaam- aangeven met een eigen pad naar die directory:

```
Set MijnProg$BestandsDir <Obey$Dir>.Bestanden
```

en wordt de regel voor de IconSprites:

```
IconSprites <MijnProg$BestandsDir>!Sprites
```

Deze wijze van aanroepen van bestanden is niet voorbehouden aan de !Boot of de !Run-file. Ook in je Basic-programma doe je dat zo:  
x%=OPENUP( "<MijnProg\$Dir>.HighScores" )

### 3.3 de !Run-file

Dit bestand wordt op dezelfde wijze gemaakt als de !Boot-file want het is ook een obey-file. Hier is het te vergelijken met een batch zoals die onder DOS en Windows worden toegepast. Je kunt er de werkomstandigheden voor jouw programma mee beïnvloeden. We zetten er de volgende vier regels in al zullen dat er in de loop van de tijd ook meer kunnen worden:

```
Set MijnProg$Dir <Obey$Dir>
WimpSlot -min 8K -max 8K
IconSprites <Obey$Dir>.!Sprites
Run <Obey$Dir>.!RunImage
```

Regel 1 en 3 zijn hierboven al verklaard.

Regel 2 geeft aan dat voor dit programma 8Kb geheugenruimte moet worden gereserveerd. Vrijwel altijd zijn de waarden achter min en max dezelfde, maar ze mogen verschillen. Bij min geef je dan aan hoeveel ruimte er minimaal nodig is in geval van geheugenkrapte. 640Kb is de standaardwaarde. Met andere woorden, als je deze regel niet zou toevoegen, reserveert RISC OS 640Kb voor jouw programma. Als dat programma maar klein is, wordt er dus teveel gereserveerd. 'MijnProg' is nu nog klein, vandaar dat we 8Kb meegeven.

Regel 4 is nodig om aan te geven wat de naam van het eigenlijke programma is dat moet worden gestart. Gangbaar is om programma's !RunImage te noemen al mag daar ook een andere naam voor gebruikt worden als het programmabestand dan ook maar die naam heeft.

Nu kunnen er in de !Run-file meer opdrachten worden gegeven. Zo kun je met RMEnsure modulenaam zorgen dat eerst gecontroleerd wordt of een bepaalde module wel aanwezig is en met RMLoad kan vervolgens die module geladen worden. Het is zelfs mogelijk nog opdrachten te geven na de Run. Die opdrachten zullen pas worden uitgevoerd als het programma wordt afgesloten. In het volgende deel van deze cursus komen deze aanvullende aspecten aan de orde.

### 3.4 een sprite-file

Ons wonderschone programma verdient natuurlijk een eigen icoon waarmee het in een directory getoond wordt of op de icoonbalk. Eigenlijk zijn het eigen iconen, want er zijn er vaak meer: !Sprites, !Sprites22 en !Sprites23. Ieder is voor een eigen doel gemaakt.

!Sprites is de oudste variant. De resolutie ervan is laag en het aantal kleuren beperkt. Ze zijn voor lage scherminstellingen (we spreken niet meer van modes) als 480x352 of 640x480 en 16 kleuren.

!Sprites22 is de nieuwste met een grotere detaillering en meer kleur. Die zijn bedoeld voor 800x600 of meer met 256 kleuren.

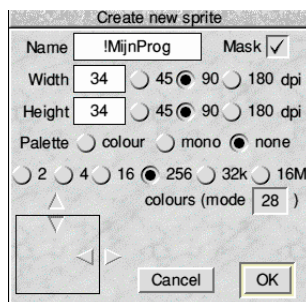
!Sprites23 is voor monochrome beeldschermen. De resolutie is hoog, maar de sprite is monochroom. Dat moet wel omdat een kleursprite op een monochroom scherm heel anders kan uitvallen dan de bedoeling is. Groen en rood kunnen daar dezelfde grijswaarden krijgen zodat het verschil wegvault.



We maken in onderstaand voorbeeld een !Sprite22. Maar om ook met de andere situaties rekening te houden, zouden we ook de andere twee sprites moeten maken. De volgende specificaties zijn van belang:

!Sprites	!Sprites22	!Sprites23
lage resolutie	hoge resolutie	hoge resolutie
34x17	34x34	34x34
16 kleuren	256 kleuren	monochroom
W90/H45 dpi	90 dpi	90 dpi
mode 12	mode 28	mode 28

In !Boot en !Run staan aangegeven waar het bestand !Sprites te vinden is. Dat hoeft niet gewijzigd te worden in !Sprites22 of -23. Het OS vindt een spritefile en zoekt, afhankelijk van de beeldscherminstelling naar -22 of -23. Vindt het dat, dan wordt een van die gebruikt en anders geldt de standaard.



Om dat icoon te maken starten we !Paint. Klik op het icoon op de iconbalk en het venster 'Create new sprite' verschijnt en een venster daarachter met de naam <untitled>. We vullen de naam in van onze sprite en die wordt !MijnProg. De naam moet dezelfde zijn als die van je programma, omdat anders niet jouw icoon wordt getoond, maar die van App. En daar hebben we niet al die moeite voor gedaan. Mask aanvinken, Width wordt 34 en Height 34, beide op 90 dpi. Palette op colour en het aantal wordt 256. Klik dan op OK en nu zijn er opnieuw twee

vensters. Een met de naam <untitled> met daarin een wit vlak met daaronder de naam !mijnprog. En een met in de titelbalk de naam !mijnprog. Die gaan we bewerken. Muispijl erop en met menu naar Zoom. Daar naar rechts en klik in het pijltje linksboven tot het groot genoeg is. Nu mag je naar hartelust tekenen. Vorderingen zijn ook zichtbaar in het scherm <untitled>.

Mask moest aangevinkt worden omdat hiermee je palet wordt uitgebreid met de kleur transparant. Zo kun je ervoor zorgen dat op sommige plaatsen in je icoon de achtergrond zichtbaar blijft.

Er is een opmerking nodig. Bestanden krijgen een icoon met een kader er omheen en bij programma's ontbreekt dat kader. Vergelijk het programma-icoon van Paint met het bestandsicoon van Sprites22. Zo zijn ze van elkaar te onderscheiden en je zou daar tijdens het tekenen rekening mee kunnen houden.

Bewaar het bestand in de programmadirectory !MijnProg onder de naam !Sprites22.

Nu heeft !Paint tamelijk veel mogelijkheden maar er zijn ook beperkingen. Je zou daarom ook een sprite kunnen maken met !Photodesk. Je kunt daar zelfs een bewerkte foto voor gebruiken.



Wie liever !Draw of !ArtWorks gebruikt, kan rustig zijn gang gaan. Alleen wordt daarmee een vectorafbeelding gemaakt, en geen bitmap of sprite. Je hebt er een conversieprogramma voor nodig als !ImageMaster, maar je kunt ook een snapshot maken met !Paint. Ook het programma !Snapper maakt het mogelijk een fotootje te maken en daarmee een sprite.

De afmetingen van 34 bij 34 zijn niet heilig. Ze mogen groter zijn maar een !Sprites is meer dan een plaatje. Er zit een klikbaar gebied achter. Dat is beperkt tot 34 x 34 en je mag een !Sprites22 zo groot maken als 'de Nachtwacht', maar de klikbaarheid neemt niet toe.

Naast de sprite voor het programma, moet er nog een tweede sprite worden gemaakt: sm!sprite. De afmeting daarvan is kleiner en de naam begint met sm. In ons voorbeeld dus sm!mijnprog. Dat kleine spriteje is voor degenen die in de Filer onder Display Small icons hebben geactiveerd. Nu kan RISC OS de normale sprite voor dit doel verkleinen, maar dan blijft er misschien te weinig van het plaatje over, vandaar dat er een alternatief plaatje gemaakt kan worden.

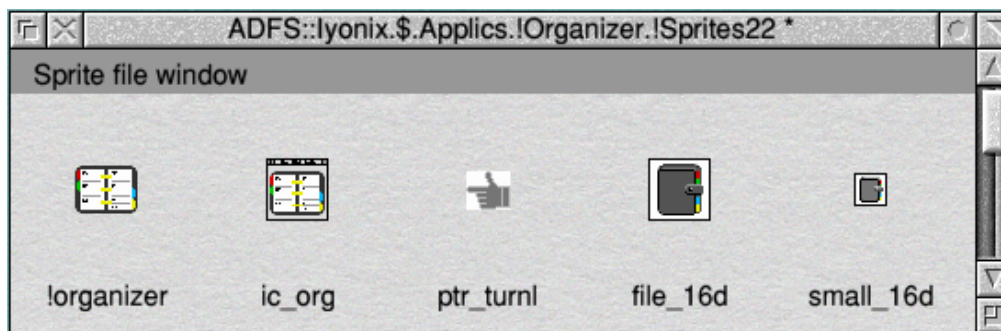


De afmetingen zijn als volgt:

!Sprites	!Sprites22	!Sprites23
17x9	17x17	17x17

de verdere specificaties zijn hetzelfde.

Je maakt die sm!sprite door de muispijl in het venster <untitled>, of, als het al een naam heeft, <!Sprites22> te zetten, daar via 'menu' te kiezen voor 'New sprite' en stap voor stap het proces opnieuw uit te voeren. Je kunt ook een bestaande sprite uit een ander programma Saven in dit venster. Zo ontstaat er een bestand (!Sprites22) met meerdere sprites (als !mijnprog of sm!mijnprog) erin. Dat bestand dient bewaard te worden in het venster met de naam !Sprites22. Zie het voorbeeld hieronder.



Naast sm!sprite zijn er nog enkele andere sprite-types. Ze zijn bedoeld voor bijzondere situaties en ik vermeld ze omdat het kan zijn dat je ze zou willen of moeten gebruiken:

file\_XXX is een sprite om een bepaald bestandstype aan te geven. Op de plaats van de drie ixen komt een hexadecimale code van drie tekens. Ze geven het bestandstype aan, we komen er in deel 2 op terug.

ic\_naam is een sprite die te maken heeft met het nieuwe knopje dat bij RISC OS 4 rechtsboven zit. Daarmee kan een van een venster een icoontje gemaakt worden dat op het pinboard wordt geprikt. Er moet een randje om de sprite gemaakt worden dat lijkt op de balken rondom een venster en het is aardig er een speldje bij te maken.



ptr\_naam is voor sprites die gebruikt worden als pointer; ze komen soms in de plaats van de muispijl. Handig om in verschillende situaties de pijl te veranderen in een pointer die iets zegt over een bepaald gebruik. Denk aan een handje, potlood, spuitbus of kruishaar. Een zandloperkje hoeft zo niet te worden gemaakt omdat dat op een andere manier kan worden geactiveerd. Ook dit komt in het vervolg van de cursus aan de orde.

small!naam mag in plaats van sm!naam gebruikt worden.

Het is wel leerzaam om de sprites te zien die bij het systeem horen of die in RAM aanwezig zijn. De volgende drie opdrachten zorgen ervoor dat er twee bestanden worden gemaakt vol met sprites: ROMSprites en RAMSprites. Activeer je editor, stel die in op Basic en voer de opdrachten in. Bewaren, activeren en bekijken maar.

```
SYS "Wimp_BaseOfSprites" TO rom%,ram%
SYS "OS_SpriteOp",268,rom%,"ROMSprites"
SYS "OS_SpriteOp",268,ram%,"RAMSprites"
```



### 3.5 de !Help-file

Het gebeurt nog steeds te vaak dat een programma geleverd wordt zonder !Help. Dat bestand kan opgevraagd worden door de muispijl op het programma-icoon te zetten, op menu te drukken en dan via App. '!naam' naar rechts te gaan, zodat een submenu verschijnt. In die lijst staat 'Help'. Door dat aan te klikken, verschijnt in een tekstvenster de inhoud van !Help. In dat bestand kan iets geschreven worden over je programma. Wat de volgorde van de onderwerpen is, maakt niet zo heel veel uit. Ik vind het zelf altijd prettig als bovenin de naam van het programma en het versienummer zichtbaar is. Verder wil ik graag weten wat het programma precies doet, zonder al te gedetailleerd te zijn. Dat alles liefst zichtbaar in het venster dat geopend is, zonder dat ik hoef te scrollen. Verderop kan een gedetailleerdere beschrijving, gebruiksaanwijzing, naam van de auteur en datum, post- of e-mailadres om te kunnen reageren en een disclaimer waarin je afstand neemt van schade die eventueel ontstaat door gebruik van jouw programma. Al met al een nuttige voorziening gezien vanuit het oogpunt van gebruikersvriendelijkheid en software-ergonomie.

Hoe maak je een !Help-file?

Start je editor en typ de tekst (van het type text). Bewaar het in je programmadirectory onder de naam !Help.

Het wordt aanbevolen om de Help-tekst te maken als HTML-pagina. Het voordeel is dat je er afbeeldingen bij kunt doen, een betere opmaak en links. De aanroep moet in de vorm van een obey-file met als naam !Help en als inhoud:

```
Set DeadLine$Dir <obey$dir>
Set DeadLine$Help <obey$Dir>.help/html
IF "<alias$@RunType_FAF>" <> "" then filer_run
<DeadLine$Dir>.help/html else filer_run
<DeadLine$Dir>.help/txt
```



### 3.6 de !RunImage

Start opnieuw je editor, maar nu met 'Create' BASIC. Typ de volgende regels:

```
PRINT "Hoi"
END
```

en bewaar dit onder de naam !RunImage in je programmadirectory. Klik nu op het icoon van je programma en kijk wat er gebeurt.

We zijn nu klaar met hoofdstuk 3. In hoofdstuk 4 zullen we de !RunImage gaan vullen met een WIMP-programma. Aan het eind van dit hoofdstuk ziet !RunImage er zo uit:

```
PRINT "Hoi"
END
```

In dit hoofdstuk staan de volgende paragrafen:

- 4.1 de SWI's 11
- 4.2 het vullen van een parameterblok 13
- 4.3 het hoofdprogramma 13
- 4.4 de WIMP-poll 14
- 4.5 het eigenlijke pollen 16
- 4.6 de parameterblokken 18

#### 4.1 de SWI's

Een programma in Basic bestaat uit een reekst opdrachten die uitgevoerd zullen worden. Daarmee kun je alles verzorgen: invoer, uitvoer, de aansturing van randapparaten of het scherm dat de gebruiker te zien krijgt. Veel van dat werk zou het OS van de programmeur kunnen overnemen. We hoeven ze dan zelf niet meer te schrijven want vrijwel ieder programma heeft met deze aspecten te maken. Er moet dan natuurlijk wel de gelegenheid zijn om die functies vanuit je programma aan te roepen. Het OS zorgt dan dat die functie uitgevoerd wordt. Omdat zo'n aanroep door het OS wordt uitgevoerd, noemen we ze interrupts. Ze onderbreken de werkzaamheden van het OS met de vraag dit klusje er even tussendoor te doen.

interrupts

We krijgen in programma's onder de WIMP al snel te maken met SoftWare Interrupts of SWI's. Die komen op verschillende plaatsen in je Basic-programma te staan. Op softwarematige wijze wordt de uitvoering van je programma even onderbroken om een stukje van het programma door het OS of door een module te laten uitvoeren. Direct na die uitvoering wordt de verwerking van je programma voortgezet tot de volgende SWI. RISC OS kent honderden mogelijkheden tot SWI. Er zijn dus ook honderden van die aanroepen; je hebt er een PRM of !StrongHelp voor nodig om te zien welke het zijn en wat ze voor je kunnen doen. Verder kan een programmeur ook zelf SWI's definiëren. Dat is echter specialistenwerk en zover zijn we in deze cursus nog niet. Er volgt nu eerst algemene uitleg over SWI's en dan gaan we ze een plaats in het programma geven.

SoftWare

Interrupts of SWI's

Wat SWI's voor je kunnen doen, moet duidelijk zijn, want je gaat er veelvuldig gebruik van maken. SWI's stellen de programmeur in staat, met een aanroep een functie te laten uitvoeren, waarmee wordt voorkomen dat de programmeur tientallen programmaregels moet schrijven om hetzelfde te laten doen. Daardoor blijven onder RISC OS programma's zo klein. Voor ons betekent het dat we met minder programmeren toch een uitstekend programma krijgen. We kunnen er zo met een SWI voor zorgen dat er bijvoorbeeld een venster op het scherm komt te staan of met een andere dat er een menu verschijnt.

Onder Basic vindt de aanroep van een SWI plaats met een SYS-call. De schrijfwijze wordt dan, algemeen gezien:

schrijfwijze

SYS "SWI\_NaamAanroep" of

SYS "nummer" maar omdat je aan dat nummer de werking van de SWI niet kunt zien, gebruiken we die liever niet.

De schrijfwijze luistert zeer nauw, aanhalingstekens, hoofd- en kleine letters en de onderstreep. Een foutje en je krijgt als foutmelding tijdens de uitvoering: 'No such SWI'.

Het eerste deel van de aanroep geeft aan waar het betrekking op heeft: ADFS, OS,

Wimp, Font, Sound etc. Dan volgt er een onderstreep en dan de functienaam.  
Twee voorbeelden uit de Wimp:

```
SYS "Wimp_Initialise"      (SWI &400C0)
SYS "Wimp_OpenWindow"     (SWI &400C5)
```

## 4.2 parameters

parameters

registernummers

Nu geven we wel aan wat er moet gebeuren, maar er moet wel bijgezegd worden hoe of waarmee dat moet. Bijvoorbeeld hoe groot het venster moet worden en welke balken of knoppen er in zitten. Daarom worden er parameters meegegeven. Die komen erachter te staan. In de beschrijving van de SWI (zie de PRM of StrongHelp) worden ze ook aangegeven in de vorm van registernummers als R0, R1 of R2. Dat kunnen er maximaal 27 zijn, maar het zijn er hooguit vijf. Je kunt aan de beschrijving ook altijd aan zien hoeveel parameters er zijn en wat erin wordt meegegeven:

```
SYS "Wimp_Initialise",R0,R1,R2,R3 TO R0,R1
SYS "Wimp_OpenWindow",,R1 (R0 wordt niet gebruikt en blijft dus open, hoewel
de komma ervoor wel vermeld moet worden. Anders leest het OS R1 op de plaats van
R0 en dan ontstaan er fouten.)
```

invoer en  
uitvoer

On entry of =>  
On exit of <=

De TO achter "Wimp\_Initialise" geeft aan dat er ook iets terugkomt. Na verwerking van zo'n aanroep komt er een resultaat in registers te staan. Er is dus sprake van invoer (voor de TO) en uitvoer (achter de TO).

Dat in- en uitvoeraspect vind je in de PRM en in !StrongHelp terug in de beschrijvingen. Daar staat bij On entry of => wat je meegeeft en bij On exit of <= wat je terugkrijgt.

Bij "Wimp\_Initialise" is dat bijvoorbeeld:

On entry

R0 = laatste versienummer van de Wimp maal 100  
R1 = het woord 'TASK' omgekeerd in hexadecimaal (&4B534154)  
R2 = een verwijzing naar een korte omschrijving van de taak waarmee die vermeld kan worden in het venster van de Task Manager; we geven de naam van het programma op.  
R3 = een verwijzing naar een lijst van berichten. (We laten dat voorlopig buiten beschouwing.)

On exit

R0 = huidige versienummer maal 100  
R1 = taakhendel (task handle), de identificatie van ons programma.

En bij "Wimp\_OpenWindow" staat:

R1 = verwijzing naar een blok.

Deze gegevens moeten in de registers komen. Daar zijn twee manieren voor: door middel van een variabele of door middel van een blok.

variabele

Eerst de variabele: We doen dat heel simpel door een variabele een waarde te geven, de naam van de variabele in plaats van de registraanduiding te zetten en dan de SWI aan te roepen:

```
versie%=4.02*100 (het hoogste versienummer van RISC OS maal 100)
app$="MijnProg"
SYS Wimp_Initialise",versie%,&4B534154,app$,R3 TO
versie%,taak%
```

Dan het blok: Nu kan een register 32 bits bevatten oftewel 4 bytes. Dat is niet veel. Als de waarde die moet worden doorgegeven daar te groot voor is, moeten we een truukje toepassen. We zetten nu de waarde niet in het register maar in een gereserveerd stuk geheugen, een array, dat eerst gedeclareerd moet worden met DIM. Nu wordt de naam van dat geheugenblok in het register geplaatst, hier in R1:

blok

```
DIM blok% 255
REM vul het blok met data
SYS "Wimp_OpenWindow" ,,blok%
```

Doe dit bovenstaande nog niet, het is alleen als voorbeeld. De data moet op strikt voorgeschreven wijze in dat blok komen te staan, omdat het OS anders niet weet welke waarde waarvoor gebruikt moet worden. We hebben er nog niet meteen mee te maken zodat we daar nu nog geen aandacht aan besteden, maar in paragraaf 4.6.

Een SWI is net een automaat: je stopt er iets in en er komt wat uit. Dat resultaat kan op twee manieren: 1 een resultaat kan zichtbaar worden, er verschijnt dan bijvoorbeeld een venster op het scherm, of 2 het resultaat komt in een register te staan en moet eerst worden geïnterpreteerd voordat we er iets mee kunnen doen.

SYS "Wimp\_OpenWindow" is een voorbeeld van de eerste. In blok% heb ik allerlei kenmerken gezet van het venster. Afmetingen, plaats op het scherm, schuifbalken, kleuren, knoppen etc zijn erin vermeld. Hoe dat moet, wordt verderop duidelijk. Het resultaat na de aanroep is het venster met die kenmerken op het scherm.

SYS "Wimp\_Initialise" is een voorbeeld van de tweede. Het resultaat komt in R0 en R1 te staan. In R0 of versie% komt de huidige RISC OS versie en in R1 (taak%) komt de identificatie van ons programma.

### 4.3 het hoofdprogramma

Basic staat ons toe procedure en functies te gebruiken. Daardoor kan er een programmastructuur worden toegepast van een (kort) hoofdprogramma met verwijzingen naar die procedures en functies. Het hoofdprogramma zal bestaan uit een inleiding, de kern en het slot en de vermelding END om aan te geven dat het programma hier echt eindigt. Ik heb er namen voor gebruikt die zo duidelijk mogelijk aangeven wat er gebeurt. Dat is in verband met deze cursus. Het staat natuurlijk ieder vrij de meer gangbare benamingen te gebruiken, ze staan er tussen haakjes achter:

hoofdprogramma  
procedures  
functies

```
PROCbegin      (PROCinit)
PROChoofd     (PROCmain)
PROCeinde     (PROCclose)
END
```

In PROCbegin zetten we alles klaar om het programma aan te melden bij het OS, we definiëren variabelen en blokken, roepen templates aan, laten schermen verschijnen, etc.

In PROChoofd bouwen we de hoofdilus waarmee ons programma en het OS contact met elkaar hebben. Hier wordt alle in- en uitvoer doorgegeven en vindt de verwerking ervan plaats.

In PROCeinde geven we aan wat er gebeuren moet als we ons programma beëindigen, want dan moet er vanalles worden afgemeld.

Open in onze programmadirectory het bestand '!RunImage' door erop te klikken met de Shift ingedrukt. Veeg de eerste regel (PRINT "Hoi") uit en zet er de bovenstaande programmaregels voor in de plaats. Bewaar dat maar vast. Maak er toch een gewoonte van je programma regelmatig te save, anders moet je al dat invoerwerk misschien opnieuw doen.

Nu werkt je programma niet meer want de procedures worden wel aangeroepen, maar zijn nog niet uitgewerkt. De foutmelding is 'No such function/procedure'.

Tussen DEF PROCbegin en ENDPROC komt alles te staan dat voor het correct starten van PROC hoofd van belang is. We laten dit gedeelte nu nog leeg, omdat we het gaan vullen vanuit de beschrijving van PROC hoofd.

De eindige lus

In DEF PROC hoofd maken we een WHILE-ENDWHILE. Bij het eerste deel van de lus, de WHILE, geven we aan onder welke omstandigheid die lus doorlopen zal worden. We geven daarmee ook aan wanneer die lus moet stoppen. 'Zolang niet klaar' kan een goed uitgangspunt zijn. WHILE klaar%=FALSE in Basic. Wordt die waarde TRUE dan zal de lus verlaten worden en moet het programma eindigen door PROC einde aan te roepen.

We moeten de variabele buiten de lus een startwaarde geven. Daarvoor hebben we PROC begin. Zet daar dus maar in: klaar%=FALSE

Tussenstand:

```
PROCbegin
PROC hoofd
PROC einde
END
```

```
DEF PROCbegin
  klaar%=FALSE
ENDPROC
```

```
DEF PROC hoofd
  WHILE klaar%=FALSE
    REM ergens wordt klaar%TRUE anders loopt deze lus
    eindeloos door
  ENDWHILE
ENDPROC
```

```
DEF PROC einde
ENDPROC
```

Het inspringen en de witregels doen we erbij om de structuur duidelijk te houden. Als je dit programma probeert, lijkt het of je systeem hangt. Een druk op de Escape-toets rukt hem echter uit de lus.

Het wordt tijd de poll te beschrijven.

#### 4.4 de WIMP-poll

Het programma tot nu toe is singletasking. Om het multitasking te maken, moeten er drie dingen gebeuren in de drie delen van ons programma. In PROC begin moet het programma worden aangemeld. Dat doen we met de opdracht:

```
SYS "Wimp_Initialise",R0,R1,R2 TO R0,R1
```

We hebben het bij het voorbeeld al gezien:

R0 vraagt om de laatste RISC OS-versie maal 100: bij mij is dat  $4.02 * 100 = 402$ .<sup>1</sup>

R1 vraagt om het woord “TASK” omgekeerd in hexadecimaal en dat is “&4B534154”. Dat moet letterlijk zo worden ingevoerd.

R2 vraagt om de naam van ons programma en dat is “MijnProg”

Na de uitvoering staan in R0 de huidige versie van het OS en in R1 de identificatie van dit programma. De laatste hebben we nodig als we het programma weer afmelden.

versie  
identificatie

Nu mag het zo:

```
SYS “Wimp_Initialise”,402,&4B534154,”MijnProg” TO R0,R1
```

Maar het is duidelijker de verschillende delen toe te kennen aan een variabele en die in de aanroep plaatsen:

```
versie%=4.02*100  
app$="MijnProg"  
SYS “Wimp_Initialise”,versie%,&4B534154,app$ TO versie%,taak%
```

Je kunt dit zo toevoegen aan PROCbegin

Nu de aanpassing in PROChoofd.

Binnen de lus moet de opdracht

```
SYS “Wimp_Poll”,R0,R1,R3 TO R0,R1,R2
```

In R0 wordt een 32-bits woord gevraagd als filter op de reason code. We hebben het er aan het eind van deze paragraaf nog even over, nu zetten we er 0 in.

In R1 moet zoveel data dat het niet in een register past en daarom zetten we er de naam in van een geheugenblok. Dat is er nog niet en daarom maken we het nu. In PROCbegin brengen we een array aan van 256 bytes met de naam blok%:

filter

```
DIM blok% 255
```

blok

De uitvoering van de aanroep SYS “Wimp\_Poll” resulteert in dat wat in R0 achter TO staat. Dat moet een variabele worden, we nemen maar reason%.

Daarin wordt vermeld wat het OS gesignaleerd heeft dat van belang is voor ons programma. Daarin kan een veelheid aan zaken worden geplaatst (wel 16) en die moeten later worden uitgezocht.

---

<sup>1</sup> Er zijn twee mogelijke uitgangspunten voor het versienummer: de huidige versie en het laagste versienummer waaronder jouw programma draait. Dat verschil is niet zo belangrijk al is de PRM er duidelijk in: het gaat om de *current RISC OS version*. De bottleneck zit in de SWI's die je gebruikt. Als er ook maar een is die onder versie 4.02 of hoger thuishoort, werkt je programma onder oudere versies niet. Niet vanwege het versienummer dat in R0 staat, maar vanwege die SWI. Zo is het versienummer te beschouwen als een datering.

Postdateren mag (vooruitdateren als 5.02) al is het niet nuttig, maar antedateren mag kennelijk ook (terugdateren als 3.05). En dan lijkt mij toch dat je de huidige datering gebruikt, de versie waaronder jij het programma gemaakt hebt.

Als je jouw programma geschikt wilt maken voor oudere RISC OS-versies, zou je moeten voorkomen dat je te nieuwe SWI's gebruikt. Of je past het volgende toe:

```
SYS “Wimp_Initialise” geeft achter TO in R0 de RISC OS-versie terug van het apparaat waarop jouw programma draait. Daarmee kun je controleren of die versie te oud is en als dat zo is, kun je een bericht aan de gebruiker sturen dat dit programma niet draait op zo'n oud systeem. Of je laat je programma een andere weg vervolgen om de nieuwere SWI's te vermijden. Dan moet de gebruiker maar accepteren dat bepaalde aspecten niet volgens de nieuwe standaards zijn uit te voeren.
```

De aanroep wordt nu:

```
SYS "Wimp_Poll",0,blok% TO reason%
```

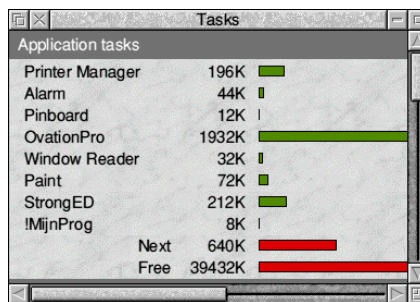
afmelden

Het derde deel moet in PROCeinde worden gedaan. Als het programma beëindigd wordt, dient het te worden afgemeld. We geven daarbij de identificatie van ons programma op en de hexcode van het omgekeerde woord TASK. Dat doen we met de opdracht:

```
SYS "Wimp_CloseDown",taak%,&4B534154
```

Breng alle bovenstaande wijzigingen aan en save je programma.

Als je het nu start, blijft je systeem gewoon doorgaan. Het kan niet meer met Escape worden gestopt.



Nu is er een achterdeurtje. Klik op de Switcher, het symbool van Acorn of RISC OS in de rechterbenedenhoek. De Taskmanager laat een scherm zien en daar zie je 'MijnProg' staan; het neemt 8k in beslag want dat heb je in de !Run-file aangegeven. Normaal zou je met de menuknop van je muis via een menuutje het programma kunnen eindigen, maar daar hebben we nog geen voorzieningen voor getroffen:

klaar% blijft FALSE en wordt niet TRUE en daardoor zal PROCeinde nooit bereikt worden.

Dat moeten we dus nu doen.

#### 4.5 het eigenlijke pollen

Om het programma te laten eindigen, moet het kunnen reageren op een actie met de muis. En als we er later een venster aan toevoegen, moet je met de muis op iconen kunnen klikken, moet het programma kunnen reageren op invoer etc. Dat kunnen we verzorgen met het pollen. Er zijn zestien punten voor ons van belang (van 0 tot 19, maar ze worden niet allemaal gebruikt) en in de loop van deze cursus komen de belangrijkste aan de orde. We noemen die punten overigens events.

We zetten de poll op met een CASE .. ENDCASE en daartussen komen de meetpunten of events.

```
CASE reason% OF
  WHEN 0:
  WHEN 1:
  ..
  WHEN 19:
ENDCASE
```

Per event kunnen we een procedure voor de afhandeling laten zorgen. Hieronder staan de events met hun namen die aangeven waar het meetpunt voor zorgt:



!SciCalc in hex converteren en dat meegeven: &1C33. Aan het einde van paragraaf 4.6 staat hoe dat moet.

Zet nu in PROCbegin:  
filter%=&1C33

en in PROC hoofd wordt de regel veranderd in:  
SYS "Wimp\_Poll",filter%,blok% TO reason%

Alleen al met bovenstaand filter kan een besparing van geclaimde tijd worden bereikt van bijna twintig procent!

#### 4.6 de parameterblokken

En dan nu de parameterblokken. We hebben er een aangemaakt met de naam blok% en een grootte van 256 bytes. Dat is voor een groot deel van de klussen genoeg. Je kunt hem namelijk steeds opnieuw gebruiken, in allerlei situaties. In een aantal gevallen heb je een groter blok nodig en soms een voor een speciale situatie, maar die kunnen altijd in PROCbegin worden aangemaakt.

Nu zijn er drie aspecten aan de inhoud van blok%. Het eerste is dat je een duidelijk beeld moet hebben van de data die erin moet en de plaats ervan in het blok. Dat beeld wordt pas duidelijk als je de PRM of !StrongHelp erop naslaat. Het tweede aspect is hoe je die data er op de juiste plaats inkrijgt. En het derde aspect is hoe je de data die RISC OS teruggeeft eruit moet halen. Kortom: wat moet erin en op welke plaats moet dat komen, hoe schrijf je dat daarin en hoe lees je de data uit.

Wat moet er in zo'n blok en op welke plaats  
We hebben een blok aangegeven van 256 bytes. We zullen zien dat de data daarin zich bytegewijs, en daarbinnen soms bitgewijs, voordoet. Om een duidelijk overzicht te krijgen van de betekenis van een gegeven en de hoeveelheid bits die ervoor gebruikt wordt, is, nogmaals, de PRM of !StrongHelp nodig.

Voorbeeld

We nemen als voorbeeld de SWI SYS "Wimp\_CreateMenu",R0,R1,R2,R3

R0 wordt niet gebruikt.

R1 bevat de verwijzing naar het blok.

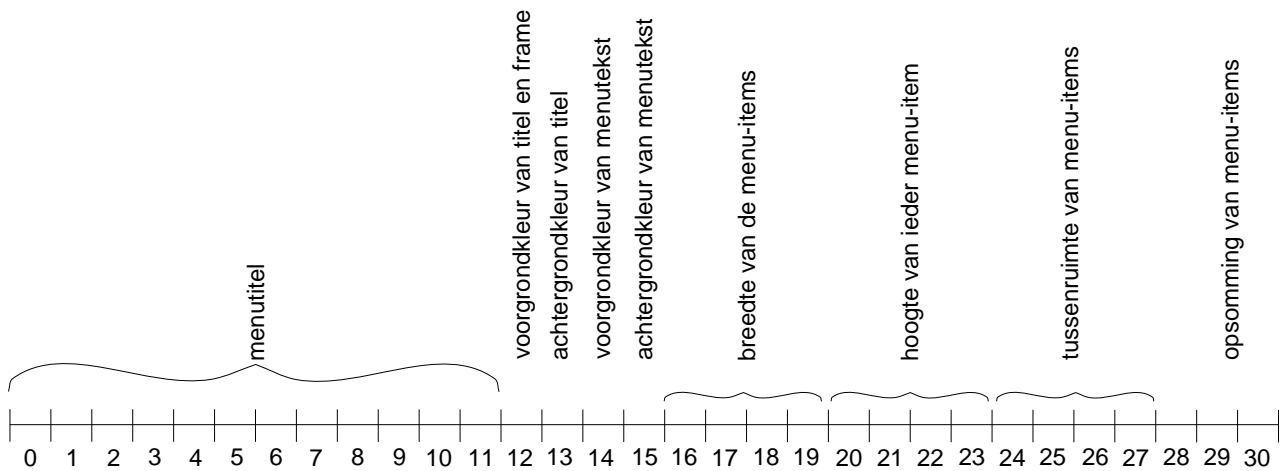
R2 bevat de x-coördinaat van de linkerbovenhoek van het menu

R3 bevat de y-coördinaat van de linkerbovenhoek van het menu.

Het gaat natuurlijk om R1: het blok; we noemen het hier menublok%. (Als we het werkelijk zo zouden gebruiken, zou het in PROCbegin gedeclareerd moeten worden met DIM menublok% 255, maar dit is slechts als voorbeeld.) De verdeling van bytes daarin is volgens de PRM:

menublok%+ 0	menutitel
12	voorgndkleur van titel en frame (meestal zwart)
13	achtergrondkleur van titel (meestal grijs)
14	voorgndkleur van menutekst (meestal zwart)
15	achtergrondkleur van menutekst (meestal grijs of wit)
16	breedte van de menu-items
20	hoogte van ieder menu-item
24	tussenruimte van menu-items
28	opsomming van menu-items (24 bytes voor ieder item)

Zonder al te veel in te gaan op de specifieke betekenissen –dat komt later in hoofdstuk 7– kunnen we wel zien dat het blok een begin heeft bij 0 en vandaaruit per byte of groepjes bytes wordt opgebouwd. Stel je dat rustig voor als een lineaal met centimeterverdeling.



Het onderdeel menutitel neemt 12 bytes in beslag, van 0 tot 11. Zo'n titel mag dus niet langer zijn dan 12 tekens.

Voor de voorgndkleur van titel en frame is een byte op positie 12. De breedte van het menu wordt met een woord van vier bytes vastgelegd tussen byte 16 en 19 en bevat een getal in vier bytes. Vanaf byte 28 worden menu-items aangegeven en is voor ieder item 24 bytes beschikbaar.

Het schrijven in een blok doen we met de constructie:

schrijven in een blok

operator(adres+pointer)=constante of  
operator(adres+pointer)=variabele

Het lezen van data uit een blok doen we met:

variabele=operator(adres+pointer)

[Operatoren] kunnen de volgende zijn:

operatoren

- ? lees of schrijf een byte (1 byte);
- ! lees of schrijf een 32 bit woord (4 bytes);
- | lees of schrijf een getal met decimalen (5 bytes);
- \$ lees of schrijf een string (0..255 bytes).

Het [adres] wordt voorgesteld als de naam van het blok, in ons voorbeeld menublok%

De [pointer] is een aanwijzer die de plaats aangeeft vanaf het begin van het blok. Er wordt een getal voor gebruikt.

Een [constante] is een waarde.

Een [variabele] is een naam waaraan een waarde is toegekend.

Als we, in voorgaand voorbeeld, de menutitel in het blok willen plaatsen vanaf positie 0, dan doen we dat door:

```
$ (menublok%+0) = "MijnProg" of
menu$ = "MijnProg"
$(menublok%+0) = menu$
```

Er staat een \$ voor menublok% omdat hier een string geschreven wordt. Je mag +0 weglaten omdat het begin van het blok is.

Als we de achtergrondkleur van titel en frame willen aanbrengen, dan doen we dat als:

```
?(menublok%+12)=7 of
kleur%=7
?(menublok%+12)=kleur%
```

Er staat een ? voor menublok% omdat hier een byte geschreven wordt.

Als we de breedte van het menu willen aangeven, doen we dat met:

```
!(menublok%+16)=96 of
breedte%=96
!(menublok%+16)=breedte%
```

Er staat een ! voor menublok% omdat hier een woord van vier bytes geschreven wordt.

Nu mag?(menublok%+12) zo geschreven worden. Maar het is gebruikelijk het te schrijven als menublok%?12 (en dan kunnen de haakjes worden weggelaten). Dit geldt echter alleen voor de operatoren ? en !, maar niet voor | en \$.

Dus:

```
?(menublok%+12)= Ok      menublok%?12= Ok
!(menublok%+16)= Ok      menublok%!16= Ok
|(menublok%+22)= Ok      menublok%|22= Fout
$(menublok%+30)= Ok      menublok%$30= Fout
```

Soms moet het gedetailleerder. Dan moeten er bits in een byte worden aangegeven en als dat gedaan is, moet de byte in het blok geplaatst worden. Bij ieder menu-item (vanaf positie 28) moeten er 24 bytes worden doorgegeven die op de volgende wijze zijn opgebouwd:

bytes 0..3 menu flags

Bit	Betekenis
0	een vinkje links van item
1	onderbroken streep onder item
2	item is beschrijfbaar
3	toon bericht
4, 5 en 6	worden niet gebruikt
7	voor het laatste item in het menu als afsluiting

alle andere bits zijn 0

---

bytes 4..7 submenu pointer (>=&8000) of window handle (1-&7FFF) of -1 indien niet gebruikt

---

bytes 8..11 menu icon flag  
bytes 12..23 menu icon data

---

Het gaat om het gedeelte van byte 0..3. Om te beginnen hebben we een byte nodig, terwijl er ruimte is voor drie. Kennelijk is dat bedoeld voor uitbreiding. De bits in die byte staan er als volgt:

23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Aangezien de bits van 23..8 niet gebruikt worden, bevatten ze nullen, voorloopnullen, en die laten we weg.

Laten we nu als uitgangspunt nemen dat we een menu maken en dat na het eerste item een onderbroken streep komt en dat we verder niets instellen. Op positie van bit 1 komt een een en op de overige plaatsen een nul:

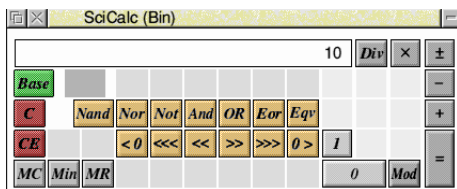
!SciCalc

7 6 5 4 3 2 1 0  
0 0 0 0 0 0 1 0

#### 4.7 converteren van binair naar hexadecimaal

We kunnen die byte binair doorgeven als %00000010 of %10 maar ook in hexadecimaal en dat is de gebruikelijke manier. Hoe wordt dat van binair in hex geconverteerd? Daarvoor zijn twee mogelijkheden: door middel van !SciCalc of handmatig.

handmatig  
nibble



Activeer !SciCalc en druk op het knopje Base tot Bin verschijnt. Dan bovenstaande byte invoeren zonder voorloopnullen. Dan op Base drukken tot Hex verschijnt en de hexcode is zichtbaar.

Bij de handmatige manier verdelen we de byte van rechts naar links in groepen van vier (nibbles). De zwaarte van iedere positie in een nibble is 8 4 2 1

%=binair  
&=hexadecimaal

$$\begin{array}{cccc|cccc}
 & 8 & 4 & 2 & 1 & & 8 & 4 & 2 & 1 \\
 0 & \leq & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \Rightarrow 2
 \end{array}$$

De zwaarte telt alleen als er een 1 onder staat. Als er meerdere enen in een nibble staan, tellen we de zwaarten bij elkaar op en dan plakken we het resultaat in de juiste volgorde weer aan elkaar: 02. Daaruit volgt dat %00000010=&02. Dat wat als resultaat in een nibble groter is dan 9, wordt voorgesteld met de bijbehorende hoofdletter: 10=A, 11=B etcetera.

Het %-teken wordt gebruikt om binair aan te geven en de & is voor hexadecimaal. Als we dit resultaat op positie 28 in menublok% willen zetten, wordt de regel:

menublok%?28=&2

Opdracht: ga na waarom bij het laatste menu-item zonder verdere toevoeging &80 ingevoerd moet worden en waarom %1110000110011 hetzelfde is als &1C33.

De laatste opmerkingen over de parameterblokken: we zijn ervan uitgegaan dat de grootte van menublok% 256 bytes is. Als je van tevoren weet hoeveel menu-items er zijn, is ook de precieze grootte van het blok te bepalen: 28+(n items\*24). Als je klakkeloos aanneemt dat 256 wel genoeg zal zijn, zal blijken dat bij tien items het blok te klein is. En dat levert een probleem op waar je nog lang naar kunt zoeken. Aan de andere kant is het ook niet nodig het blok groter te maken dan strikt noodzakelijk.

Hier eindigt dit hoofdstuk. !RunImage ziet er nu zo uit:

```
PROCbegin  
PROChoofd  
PROCeinde  
END
```

```
DEF PROCbegin  
  DIM blok% 255  
  klaar%=FALSE:versie%=4.02*100  
  app$="MijnProg":filter%=&1C33  
  SYS "Wimp_Initialise",versie%,&4B534154,app$ TO versie%,taak%  
ENDPROC
```

```
DEF PROChoofd  
  WHILE klaar%=FALSE  
    SYS "Wimp_Poll",filter%,blok% TO reason%  
    CASE reason% OF  
      WHEN 17,18:IF blok%!16=0 THEN klaar%=TRUE  
    ENDCASE  
  ENDWHILE  
ENDPROC
```

```
DEF PROCeinde  
  SYS "Wimp_CloseDown",taak%,&4B534154  
ENDPROC
```

Dit hoofdstuk bevat de volgende paragrafen:

- 5.1 een beeld van een venster 23
- 5.2 de template-editor 24
- 5.3 de vereniging van template en programma 24
- 5.4 de praktijk 27
- 5.5 een eenvoudige foutprocedure 28

Tot nu toe doet ons programma niet veel. Als we verwachten dat er iets zichtbaar is, verwachten we eigenlijk een venster. En daar gaan we nu aandacht aan besteden.

vensters

Om een venster in je programma aan te brengen, zullen er verschillende dingen moeten gebeuren. Eerst moet je je een duidelijk beeld vormen van dat venster, dan moet dat beeld worden gerealiseerd in data (die uit dat beeld moet worden gedestilleerd) en tenslotte moet die data (via een blok) in je programma worden gezet zodat het venster zichtbaar gemaakt kan worden. De details ervan worden verderop duidelijk.

### 5.1 een beeld van een venster

We moeten ons kunnen voorstellen dat we een stelsel aan het opbouwen zijn dat uit drie aspecten bestaat. Aan de ene zijde is daar de gebruiker. Dat ben jij nu. Aan de andere zijde is er het programma. Daar zijn we mee bezig. En daartussen bevindt zich het venster. We noemen het ook het gebruikersinterface: een laag tussen gebruiker en programma dat voor het invoeren en uitvoeren zorgt. Een dashboard, een balie, een plek van contact tussen mens en techniek. Daar moet behoorlijk over nagedacht worden want zo'n venster laat zien wat het programma allemaal kan zonder dat er een overdaad of een wirwar is aan knoppen, iconen en velden. Het vereist doordachtheid en structuur en het moet rekening houden met de functionaliteiten van je programma en de manier waarop de gebruiker daarmee omgaat.



Een goede manier om dat te bereiken is het venster maar eerst eens op papier te zetten. Afmetingen van het venster en van het werkveld, schuifbalken, knoppen op de drie hoeken, icons, kleuren of teksten kunnen allemaal aangegeven worden. Ook de volgorde van de invoer- of uitvoervelden, kaders om groepen van velden visueel te verzamelen en dergelijke, kunnen worden bepaald.

Nu is er wel de mogelijkheid om alle gegevens van het venster via een ingewikkelde manier handmatig in data te vertalen en die data in een blok te plaatsen, we hebben dat aan het eind van vorig hoofdstuk voor een klein deel al gezien. Hier gaat het dan om een groot aantal bytes en het vereist veel ervaring om dat ineens goed te doen. Kijk maar eens in de verschillende naslagwerken bij "Wimp\_CreateWindow". We hebben echter een hulpmiddel en dat is de template-editor. Daarmee kunnen we de tekening van ons venster via die editor in een bestand om laten zetten en dat bestand kan gebruikt worden om het blok te vullen met de desbetreffende data.

## 5.2 de template-editor

!TemplEd

We gaan !TemplEd van Dick Alstein gebruiken. Dat is freeware en het wordt geleverd op de cd bij RISC OS 4 op \$.FREESOFT.CODING.TOOLS.TEMPED134/ARC of zoek het op via Google als !TemplEd Alstein.

Het bevat goede documentatie en een !StrongHelp-bestand en het zorgt ervoor dat wij over hetzelfde kunnen praten. Activeer dus !TemplEd en zoek in verschillende programmadirectories naar template-bestanden om te bekijken hoe ze in elkaar zitten.

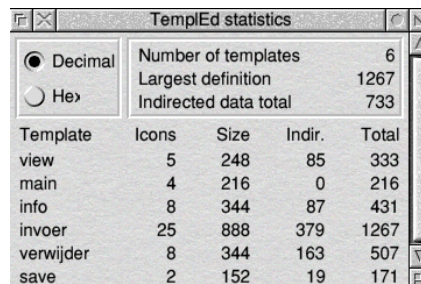
Want een andere manier om aan een goed vormgegeven venster te komen, is om een bestaande template te nemen en die voor jouw doel te bewerken. Je vindt templates in programmadirectories van veel programma's. Een verzameling voorbeelden is te vinden achter Apps op de icoonbalk, daar Open '\$', Resources en daar in diverse directories, maar ook in !TemplEd.Extras.StyleGuide. Daar zijn ook twee teksten bij, NewIcons en ReadMe, die het gebruik van templates en icons nader verklaren. Ik kan het aanbevelen om in ieder geval de !Help en de Manual te lezen die in de directory !TemplEd staan, dan hoef ik geen delen van de gebruiksaanwijzing in deze lestekst op te nemen.

## 5.3 de vereniging van template en programma

Om een template te kunnen gebruiken, moet de data ervan tijdens het laden van de template in een eigen geheugenblok worden gezet. De grootte van dat blok wordt bepaald door de hoeveelheid data. Die is van tevoren van iedere template te bepalen, tijdens het maken van de template al. Door bij !TemplEd met 'menu' om de 'statistics' te vragen, wordt van iedere template aangegeven hoeveel bytes die template groot is.

statistics

Met een vuistregel is dat, ruwweg, ook te bepalen: 88 bytes voor het venster, 32 voor ieder icon en een hoeveelheid voor ieder icoon dat indirected data bevat. Met name dat laatste leidt tot twee vragen: wat is indirected data en hoeveel bytes geheugenruimte neemt dat in beslag.



Template	Icons	Size	Indir.	Total
view	5	248	85	333
main	4	216	0	216
info	8	344	87	431
invoer	25	888	379	1267
verwijder	8	344	163	507
save	2	152	19	171

indirected data

Wat is indirected data?

We kennen drie soorten iconen: text only, text en sprites en sprites only.

In een situatie hebben we niet met indirected data te maken: bij text only, waarbij de tekst die in een venster wordt weergegeven, niet groter is dan elf bytes. Eigenlijk twaalf, maar de tekst wordt afgesloten met een extra byte in de vorm van een Return (ASCII-waarde 13 of &0D). Is de tekst langer, of is er sprake van text en sprite of sprite only, dan is de hoeveelheid data groter dan twaalf bytes. En daarvoor moet apart geheugenruimte gereserveerd worden en in die elf bytes moet nu een verwijzing (pointer) komen te staan naar dat geheugendeel met die data.

Als we hiermee te maken krijgen, en dat is al snel zo, moeten we dus bepalen hoeveel bytes indirected data er is en dan is er wat rekenwerk nodig om de grootte te bepalen van het hele blok. Dat alles is echter niet nodig om te berekenen als we de grootte laten bepalen door !TemplEd en met 'statistics' zichtbaar maken.

Stel je voor dat !TemplEd van een template laat zien dat het 245 bytes groot is. Dan moeten we dus een array maken van die grootte en met een eigen naam:

```
DIM templateA% 245
```

Om die templatefile te laden en het venster op het scherm te krijgen, moeten we vijf dingen doen:

- open het templatebestand
- laad de template
- creëer het venster } herhaal dat voor het aantal templates dat gebruikt moet worden.
- sluit het templatebestand
- vraag vensterspecificaties op en open het venster of de vensters.

Dit alles gebeurt in de initialisatie, dat we PROCbegin hebben genoemd met de volgende opdrachten (de uitwerkingen ervan staan in paragraaf 5.4 de praktijk):

```
SYS "Wimp_OpenTemplate",R1
```

waarbij:

```
R1 "<pad>.naam" waarbij <pad>.naam worden voorgesteld als  
<MijnProg$Dir>.Templates
```

open  
templatebestand

```
SYS "Wimp_LoadTemplate",R1,R2,R3,R4,R5,R6
```

waarbij ingevuld wordt:

R1 blok%

R2 naam van templatearray

R3 pointer naar einde templatearray

R4 een array van 256 bytes voor fonts of -1 voor geen fonts

R5 naam van het template

R6 voorlopig een nul

laad template

In R1 is er opnieuw de verwijzing naar het blok (blok%). We hebben het al gedeclareerd in het vorige hoofdstuk.

In R2 wordt aangegeven wat de naam is van het blok voor het venster (templateA%) en in R3 wordt verteld hoe groot dat blok is, zodat bij de aanroep het begin en het eind van dat blok bepaald kunnen worden. We mogen dat in R3 aangeven door de naam+grootte op te geven met: templateA%+245 (vooropgesteld dat het 245 bytes groot is). Nu komt er een moment dat de data van de template in blok% zal komen te staan. Als het template-array groter is dan het blok%, moet blok% vergroot worden tot de grootste waarde van dat template-array en in die situatie passen we DIM blok% aan.

Aangezien je meerdere templates mag en kunt opgeven, dient iedere template zijn eigen blok te krijgen, met een eigen naam. templateA%, templateB% etcetera zijn wel erg doeltreffend, maar het zou veel duidelijker zijn daar treffendere namen aan te geven, bijvoorbeeld de namen van de templates zelf als 'wmain', 'wsave' of 'winfo', waarbij de w ons alleen maar aangeeft dat het om een window gaat.

Bij R4 geef je het array alleen op als je in je programma andere fonts wilt gaan gebruiken dan het standaard ingestelde font. We komen hier later op terug, nu vullen we er -1 in.

Voor de naam van het template in R5 moet eerst een array gedeclareerd worden van 11 bytes. We noemen die maar even 'naam1%'. De opdracht wordt dus DIM naam1% 11 en komt bij de andere declaraties van arrays te staan.

Dan moet de naam van het te laden template nog worden toegekend aan het array:  
\$naam1%="vensterA"

En nu moet bij R5 de naam van het array worden vermeld: naam1%

Voor iedere template die geopend wordt, moet opnieuw een dergelijk array met een eigen naam worden gedeclareerd en gevuld. Dus:

DIM naam1% 11, naam2% 11  
 \$naam1%="vensterA":\$naam2%="vensterB"  
 en de IRQ "Wimp\_LoadTemplate" moet voor iedere template apart worden aangeroepen, met zijn eigen naam in R5.

Wat er in R6 komt te staan, laten we voorlopig ook buiten beschouwing, we zetten er nu nul (0) in.

creëer het venster   SYS "Wimp\_CreateWindow",,R1 TO R0  
 waarbij:  
 R1 blok%  
 R0 window handle: de SWI geeft hier een waarde terug die te beschouwen is als identificatie van dat venster. Geef hier een herkenbare naam aan als hmain%, hsafe% of hinfo%, waarbij de h ons alleen maar aangeeft dat het om een handle gaat.

sluit  
 templatebestand   SYS "Wimp\_CloseTemplate"  
 zonder parameter.

open venster(s)   SYS "Wimp\_GetWindowState",,R1 en SYS "Wimp\_OpenWindow",,R1  
 waarbij:  
 R1 blok% waarbij in blok% alle aspecten van het venster moeten komen die toegekend zijn aan de desbetreffende handle. Willen we 'main' openen, dan vullen we het blok met main door: !blok%=hmain%. Dat wordt toegekend aan "GetWindowState" zodat het met "Wimp\_OpenWindow" op de juiste plaats op het scherm kan worden geprojecteerd.

Nu staat het venster in theorie wel op het scherm, maar de bezieling ontbreekt nog. We kunnen het niet verplaatsen, vergroten, verkleinen, invullen of aanklikken, want de levendigheid kan alleen worden bereikt in de poll-lus. Daar is op plaats 2 de Wimp\_OpenWindow. RISC OS stelt steeds vast wat de toestand van ons venster is en geeft via 2 de actuele stand door aan ons programma. Daar moet het venster op grond van de nieuwe gegevens herschreven worden. We doen dat met:

```
WHEN 2:SYS "Wimp_OpenWindow",,blok%
```

Wat geopend is, moet ook gesloten kunnen worden en dat kan met:

```
WHEN 3:SYS "Wimp_CloseWindow",,blok%
```

En nu kunnen we goed zichtbaar maken wat het OS voor ons programma kan doen. In ons venster zit linksboven een sluitknop [X]. We kunnen hiermee het programma op een elegantere wijze afsluiten als we die knop kunnen gebruiken. Het OS signaleert dan de combinatie van:

- muispijl staat op de sluitknop van het venster van ons programma
- de gebruiker drukt op de linkermuisknop (klikt).

en geeft in de poll door aan positie 3 dat het venster gesloten moet worden. Als je de regel uitbreidt met:

```
WHEN 3:SYS "Wimp_CloseWindow":klaar%=TRUE
```

dan wordt tevens ons programma beëindigd. Maar bedenk wel dat dit in deze situatie, met een venster, mogelijk is; met meerdere vensters moet wel het venster gesloten worden, maar mag het programma natuurlijk niet stoppen. Verderop in de cursus vinden we daar andere mogelijkheden voor.

## 5.4 de praktijk

Nu gaan we, om bovenstaande te testen, twee vensters maken. De ene met de naam main en de andere met de naam info. Klik bij wijze van voorbeeld met menu op een willekeurig programma-icoon op de balk en kijk daar bij info.

About this program	
Name	Writer
Purpose	Word Processor
Author	© Icon Technology Ltd, 1999
Version	1.01 (02-Jul-99)

Er wordt dan een venster geopend

waarin de naam van het programma, het doel, de naam van de auteur en de versie (met datum). Maak een dergelijk venster met !TempEd en bewaar het onder de naam 'info'.

Dan het venster main. Maak een venster met verschillende iconen, het doet er op dit moment nog niet zoveel toe welke, en bewaar dat ook. Het is hier wel van belang dat je de sluitknop (linksboven) laat zitten. Vergeet niet van beide templates de statistics op te vragen want die informatie hebben we dadelijk nodig. Vergeet ook niet het geheel te bewaren onder de naam Templates in onze programmadirectory.

De aanroep van beide templates.

Declareer twee arrays met de namen van de templates en de grootte uit 'statistics'. Declareer ook twee namen van de templates met de grootte 11.

```
DIM wmain% xxx, winfo% yyy, naam1% 11, naam2% 11
```

Zorg dat op de plaats van de xxx en yyy de juiste waarden komen te staan. Als het array van blok% niet groot genoeg is, maak dan ook dat array zo groot als de grootste van xxx of yyy. Anders krijg je foutmeldingen.

Roep beide templates aan in PROCbegin, onder de DIM en na de IRQ SYS "Wimp\_Initialise", met de volgende opdrachten:

```
SYS "Wimp_OpenTemplate", , "<MijnProg$Dir>.Templates"  
$naam1%="main":$naam2%="info"  
SYS "Wimp_LoadTemplate", , blok%, wmain%, wmain%+xxx, -1, naam1%, 0  
SYS "Wimp_CreateWindow", , blok% TO hmain%  
SYS "Wimp_LoadTemplate", , blok%, winfo%, winfo%+yyy, -1, naam2%, 0  
SYS "Wimp_CreateWindow", , blok% TO hinfo%  
SYS "Wimp_CloseTemplate"  
!blok%=hmain%:SYS "Wimp_GetWindowState", , blok%  
SYS "Wimp_OpenWindow", , blok%  
!blok%=hinfo%:SYS "Wimp_GetWindowState", , blok%  
SYS "Wimp_OpenWindow", , blok%
```

Zet in PROchoofd:

```
WHEN 2:SYS "Wimp_OpenWindow", , blok%  
WHEN 3:SYS "Wimp_CloseWindow", , blok%:klaar%=TRUE
```

Bewaar het geheel voordat je controleert of het werkt.

Het is overigens wel aardig om te zien wat er gebeurt als je een aanroep maskeert met een REM ervoor. De desbetreffende regel wordt dan niet meer uitgevoerd en het is erg leerzaam om te zien wat de foutmelding of de consequentie ervan is. REM's

## 5.5 een eenvoudige foutprocedure

Er is nog een probleem. Je kunt het programma invoeren, wijzigen en testen, maar regelmatig zal blijken dat er een of meer fouten inzitten. Dat is niet zo'n probleem als de foutmelding maar duidelijk en doelmatig is. Tot nu toe hebben we daar geen aandacht aan besteed. Het komt ook eigenlijk pas in hoofdstuk 8 aan de orde, maar we hebben het nu al nodig. Daarom de volgende programmaregels voor een foutmelding.

Bovenaan het programma, op een van de eerste regels, moet komen te staan:

```
ON ERROR PROCerror:END
```

Er wordt een procedure gemaakt en die zetten we onderaan het programma:

```
DEF PROCerror
  !blok%=ERR
  $(blok%+4)=REPORT$+" (in regel "+STR$ ERL+")"+CHR$0
  SYS "Wimp_ReportError",blok%,1,app$
ENDPROC
```

!Reporter

Nu is er het programma !Reporter van Martin Avison (opzoeken via Google). Ook dat kan bij het opsporen van fouten goede diensten bewijzen. Je kunt daarvoor op verschillende plaatsen in je programma een meetpunt aanbrengen met de opdracht \*Report [parameter]. Als je gaat testen, moet het programma !Reporter geactiveerd zijn. En nu komt in een apart scherm te staan waar je om gevraagd hebt. Die vraag kan op de volgende aspecten betrekking hebben:

Tekst, OSVariabelen, BasicVariabelen en BasicExpressies. Wij gebruiken vooral de Basicvariabelen. Voer aan het eind van PROCbegin maar in \*Report app\$, naam1%, naam2%, klaar%, versie%

Nu kun je zien of de waarden kloppen. We komen er, ook wat Reporter betreft, nog op terug.

Als je bovenstaand programma hebt ingevoerd en bewaard, kun je testen om te zien of het werkt en als dat zo is, zie je twee vensters verschijnen, dat van main en info. Nu is die tweede niet zo nodig. Nog niet. We halen hem daarom niet weg, maar we verbergen hem door er REM voor te zetten:

```
REM !blok%=hinfo%:SYS "Wimp_GetWindowState",,blok%
REM SYS "Wimp_OpenWindow",,blok%
```

En zie wat er gebeurt.

Aan het eind van dit hoofdstuk ziet !RunImage er als volgt uit:

```
ON ERROR PROCerror:END
```

```
PROCbegin  
PROChoofd  
PROCeinde  
END
```

```
DEF PROCbegin  
  DIM blok% 255, wmain% xxx, winfo% yyy, naam1% 11, naam2% 11  
  klaar%=FALSE:versie%=4.02*100:app$="MijnProg":filter%=&1C33  
  SYS "Wimp_Initialise",versie%,&4B534154,app$ TO ,taak%  
  SYS "Wimp_OpenTemplate",,"<MijnProg$Dir>.Templates"  
  $naam1%="main":$naam2%="info"  
  SYS "Wimp_LoadTemplate",,blok%,wmain%,wmain%+xxx,-1,naam1%,0  
  SYS "Wimp_CreateWindow",,blok% TO hmain%  
  SYS "Wimp_LoadTemplate",,blok%,winfo%,winfo%+yyy,-1,naam2%,0  
  SYS "Wimp_CreateWindow",,blok% TO hinfo%  
  SYS "Wimp_CloseTemplate"  
  !blok%=hmain%:SYS "Wimp_GetWindowState",,blok%  
  SYS "Wimp_OpenWindow",,blok%  
  REM !blok%=hinfo%:SYS "Wimp_GetWindowState",,blok%  
  REM SYS "Wimp_OpenWindow",,blok%  
ENDPROC
```

```
DEF PROChoofd  
  WHILE klaar%=FALSE  
    SYS "Wimp_Poll",filter%,blok% TO reason%  
    CASE reason% OF  
      WHEN 2:SYS "Wimp_OpenWindow",,blok%  
      WHEN 3:SYS "Wimp_CloseWindow",,blok%:klaar%=TRUE  
      WHEN 17,18:IF blok%!16=0 THEN klaar%=TRUE  
    ENDCASE  
  ENDWHILE  
ENDPROC
```

```
DEF PROCeinde  
  SYS "Wimp_CloseDown",taak%,&4B534154  
ENDPROC
```

```
DEF PROCerror  
  !blok%=ERR  
  $(blok%+4)=REPORT$+" (in regel "+STR$ ERL+" )"+CHR$0  
  SYS "Wimp_ReportError",blok%,1,app$  
ENDPROC
```

In het volgende hoofdstuk gaan we de muis en iconen gebruiken.



Dit hoofdstuk bevat de volgende paragrafen:

- 6.1 de muis 31
- 6.2 uitvoer in iconen 33
- 6.3 invoer uit iconen 35

### 6.1 de muis

We hebben nu een programma met vensters. Maar in die vensters zitten geen klikbare iconen en ook geen iconen voor toetsenbord invoer of uitvoer; we hebben er nog niet specifiek voor gezorgd. Toch vormen die elementen de mogelijkheden om te communiceren met ons programma. Die moeten we er dus in zetten en we moeten zorgen dat ons programma gaat reageren op dat wat wij met muis of toetsenbord doen. Het is misschien een hele geruststelling om te zeggen dat het meeste door RISC OS bijgehouden en afgehandeld wordt; of de muispijl op ons venster zit, of ons venster actief is, of de pijl op een icoon staat, welke muisknop of toetsenbordtoets we indrukken, het wordt gesignaleerd en via de Wimp-poll doorgegeven. Wij moeten op de juiste plaats, bij de events, de boodschappen in ontvangst nemen en zorgen dat ons programma op de juiste wijze gaat reageren.

We hebben daarvoor:

WHEN 6: Mouse\_Click en

WHEN 8: Key\_Pressed

Voor de aardigheid kunnen we ook gebruik maken van:

WHEN 4: Pointer\_Leaving\_Window en

WHEN 5: Pointer\_Entering\_Window

mouse click  
key pressed

pointer leaving  
window  
pointer entering  
window

We passen de laatste even toe, bij wijze van test. Zet in je programma maar eens op de juiste plaats:

WHEN 4: SOUND 1, -10, 80, 4

WHEN 5: SOUND 1, -10, 200, 4

Bewaar dat en start je programma. Als je de muis over de vensters beweegt, hoor je het effect als het filter dat niet tegenhoudt. Zet daarvoor in PROCbegin filter% even op 0. Je zou beide regels nu ook meteen weer kunnen verwijderen, want de boodschap is wel duidelijk.

Om de muisklik op de juiste wijze te interpreteren, moeten we in ieder geval drie dingen weten over de toestand van de muis: de muisknop die wordt ingedrukt, het venster waarin dat gebeurt en het icoon in dat venster. Via event 6 wordt de volgende informatie in het blok geplaatst:

blok%+0 x-positie van de muispijl  
4 y-positie  
8 muisknop  
12 venster handle  
16 icoon handle

x- en y-positie

muisknop  
venster  
icoon

Als we dan de data uit het blok halen met opdrachten als:

x%=blok%!0: y%=blok%!4: knop%=blok%!8

venster%=blok%!12: icoon%=blok%!16

dan kan met die data elders iets gedaan worden.

We kunnen de data ook in een procedure doorgeven via de vijf parameters:

```
WHEN 6:PROCmuis(blok%!0,blok%!4,blok%!8,blok%!12,blok%!16)
```

De procedure wordt uitgewerkt met:

```
DEF PROCmuis(x%,y%,knop%,venster%,icoon%)  
  REM Wat te doen met de data die nu in x%, y%, knop%, venster% en  
  icoon% zit?  
ENDPROC
```

x- en y-positie	Wat we er verder mee doen, is afhankelijk van hetgeen je programma doet. Als het van belang is de schermposities van de muispijl op te vragen, kun je de waarden in x% en y% gebruiken.
muisknop	Als je wilt weten welke muisknop wordt ingedrukt, gebruik je de waarde in knop%. Dat kunnen minstens vier waarden zijn: 0 voor geen actie, 1 voor rechters (pas-aan of adjust), 2 voor middelste (menu) en 4 voor links (selecteer of klik). De andere komen later aan de orde. Hou de opmerkelijke eigenschap in de gaten: links=4, midden=2 en rechts=1.
vensters	Gebruik je meerdere vensters, dan kan met venster% bepaald worden welke handle van toepassing is.
iconen	Staan er meerdere iconen in je venster, ieder met een eigen nummer, kun je met icoon% nagaan op welk icoon de gebruiker geklikt heeft. Het is daarvoor zaak bij het maken van je template goed bij te houden welke nummers je iconen gekregen hebben.

De inhoud van venster.icoon.knop bepaalt de actie die erop volgt in de vorm van:  
IF (venster%=dit) AND (icoon%=dat) AND (knop%=zus) doe PROCzo  
maar het kan ook als:

```
CASE (venster%=dit) AND (knop%=dat)  
  CASE icoon% OF  
    WHEN zus: PROCzo  
    WHEN broer: PROCzozo  
  ENDCASE  
ENDCASE
```

In ons programma wordt dat:

```
CASE (venster%=hmain%) AND (knop%=4)  
  CASE icoon% OF  
    WHEN 11:klaar%=TRUE:REM icoon 11 is een knopje met Cancel  
  ENDCASE  
ENDCASE
```

Merk op dat bij AND en OR de expressies links en rechts ervan tussen haakjes gezet worden.

Als het icoon voor afdrukken nummer 10 is, dan wordt het:

```
WHEN 10:PROCprint  
en PROCprint zet de afdrukpoort open met VDU2, we laten de waarden van de  
variabelen afdrukken en met VDU3 wordt de poort weer gesloten.
```

Als het icoon voor bewaren nummer 9 is, dan wordt het:

```
WHEN 9:PROCbewaar  
en PROCbewaar opent het bestand, we laten de waarden van de variabelen  
weschrijven en het bestand wordt weer gesloten.
```

Als icoon 5 zo'n driehoekje is, de punt naar beneden, dan wordt het:

```
WHEN 5:waarde%=waarde%-1
```

Het is daarbij zaak de constructie zo in elkaar te zetten dat de afhandeling efficiënt gebeurt en dat ons schrijfwerk beperkt blijft. Daarover later meer. Kijk ook eens bij andere programma's hoe dat daar is opgelost en probeer te begrijpen wat daar staat.

Dit is bijvoorbeeld ook mogelijk:

```
CASE (knop%=4) AND icoon% OF
WHEN 1: PROCdit
WHEN 2: PROCdat
WHEN 3: IF venster%=hsave% PROCzus
ENDCASE
```

Als we nu eerst de manier onderzoeken waarop data (de waarde van een variabele) in een icoon in een venster worden geplaatst, dan kunnen we een venster maken waarin alle muisaspecten zichtbaar gemaakt kunnen worden.

## 6.2 uitvoer in iconen

Om iets in een icoon te plaatsen, moeten we er –ook hier– drie dingen van weten:

welk venster,  
welk icoon,  
welke waarde.

venster  
icoon  
waarde

Die drie aspecten moeten in het blok geplaatst worden en dat blok moet worden doorgegeven in een SWI. Eigenlijk zijn er twee SWI's nodig. Dat komt omdat we eerst kenmerken van een icoon moeten opvragen, voordat we er iets in kunnen zetten. Dus eerst SYS "Wimp\_GetIconState" en dan SYS "Wimp\_SetIconState". GetIconState vraagt om een pointer naar een blok met daarin op plaats 0 de naam van de window handle en op 4 het nummer van het icoon. Na de aanroep staat in het blok vanalles over het icoon, maar nu volstaan we ermee dat vanaf byte 16 icoonvlaggen staan (waar we later wat mee gaan doen) en vanaf byte 28 er de data in staat, dat wat er in ons icoon geschreven staat.

Let wel, RISC OS zet in het blok de huidige inhoud van het icoon en wij kunnen de huidige inhoud vervangen door de meest actuele inhoud.

huidige inhoud  
actuele inhoud

Dat moet in de vorm van een string. We moeten de data dus eerst in een string converteren en die string plaatsen vanaf byte 28. Vervolgens moeten er twee icoonvlaggen gezet worden. Dan roepen we SetIconState aan dat ervoor zorgt dat de string in het icoon werkelijk zichtbaar wordt.

Aangezien je dat voor meer iconen in meer vensters zou willen doen, is het verstandig daar een procedure voor te maken. Die kun je in vergelijkbare gevallen altijd gebruiken. De aanroep ervan krijgt drie parameters: venster, icoon en tekst. Eigenlijk moet die laatste 'waarde' zijn want ook een getal moet doorgegeven kunnen worden, al moet daar een string van worden gemaakt. De aanroep om in icoon 2 van venster 'hmain' de waarde van x% door te geven wordt:

```
PROCicontekst(hmain%, 2, STR$(x%))
```

We roepen de procedure in het algemeen aan met  
PROCicontekst(naam%,icon%,STR\$(text%))

In de aanroep van de procedure maken we van de laatste parameter eerst een string met  
text\$=STR\$(text%)

Vervolgens zetten we in de procedure de inhoud van naam% (de handle van het venster) in blok%!0 en de inhoud van icon% in blok%!4. We roepen de GetIconState aan en de inhoud van het blok wordt aangepast met de huidige informatie over dat icoon. Dan zetten we de tekststring in blok%!28.

In het blok moeten op byte 8 en 12 de waarden aangeven voor EOR word en Clear word. Daarmee kun je aangeven dat het type icoon moet veranderen in bijvoorbeeld een ingedrukt knopje of dat iets grijs moet worden gemaakt omdat die keuze tijdelijk niet van toepassing is. We besteden er later aandacht aan, maar nu plaatsen we er een nul waardoor aan het uiterlijk van het icoon niets verandert.

Tot slot roepen we de SetIconState aan die de actuele waarde in het icoon plaatst.

De procedure:

```
DEF PROCicontekst(naam%, icon%, text$)
!blok%=naam%:blok%!4=icon%
SYS "Wimp_GetIconState", ,blok%
$blok%!28=text$
blok%!8=0:blok%!12=0
SYS "Wimp_SetIconState", ,blok%
ENDPROC
```

Nu kun je vanuit iedere plaats van je programma zorgen dat iconen gevuld worden met een actuele waarde.

We wilden zorgen dat in een venster alle muisaspecten zichtbaar gemaakt zouden worden. Daar gaan we nu mee verder.

iconen van type:  
Data  
writeable

Laten we eerst even opmerken dat er drie typen iconen zijn waarin de waarde van een variabele kan worden doorgegeven: het type Data met validation R5, een writeable icon met validation Ktar;Pptr\_write en een writeable icon met validation Ktar;Pptr\_write;A0-9.\-

In !TempEd staan ze weergegeven als rechthoeken met daarin achtereenvolgens Data, leeg met witte achtergrond en een met -999.99

Omdat we waarden willen doorgeven die we toch niet kunnen veranderen, nemen we type Data als uitgangspunt.

Pas het venster 'main' zo aan dat het volgende zichtbaar is:

```
X-positie [          ]
Y-positie [          ]
Muisknop  [          ]
Venster   [          ]
Icoon     [          ]
```

Natuurlijk moeten de iconen recht onder elkaar staan, de namen schrijven we met een hoofdletter en worden rechts uitgelijnd. De inhoud van de data-iconen komen rechts uitgelijnd te staan, zoals dat gebruikelijk is met getallen.

Maak de breedte van de data-iconen iets groter want in die van Venster komt een grote waarde te staan.

Daaronder mogen nog wat iconen, waaronder die met Cancel want we gaan dat icoon gebruiken om het programma mee te eindigen.

Ieder icoon in ons venster krijgt een nummer en het zou handig zijn als de velden achter X-positie etcetera genummerd zijn van 0 tot 4. Als ze dat niet zijn, kun je dat met hernummeren bereiken. Vergeet niet de statistics op te vragen, in het programma de verschillende arrays een hogere waarde te geven (waarbij die van blok% zo groot moet worden als de grootste van de vensters) en de nieuwe situatie te bewaren.

In het programma zorgen we in PROChoofd voor de regel:

```
WHEN
6:PROCmuis(blok%!0,blok%!4,blok%!8,blok%!12,blok%!16)
```

En onderaan voegen we de volgende procedures toe:

```
DEF PROCmuis(x%,y%,knop%,venster%,icoon%)
  IF (venster%=hmain%) AND (knop%=4) AND (icoon%=11) THEN klaar%=TRUE
  SOUND 1,-10,(x%/6),1:SOUND 1,-10,(y%/6),1
  PROCicontekst(hmain%,0,STR$(x%))
  PROCicontekst(hmain%,1,STR$(y%))
  PROCicontekst(hmain%,2,STR$(knop%))
  PROCicontekst(hmain%,3,STR$(venster%))
  PROCicontekst(hmain%,4,STR$(icoon%))
ENDPROC
```

```
DEF PROCicontekst(naam%,icon%,text$)
  !blok%=naam%:blok%!4=icon%
  SYS "Wimp_GetIconState",,blok%
  $blok%!28=text$
  blok%!8=0:blok%!12=0
  SYS "Wimp_SetIconState",,blok%
ENDPROC
```

Let vooral op de tweede regel van PROCmuis. Daar staat icoon 11 maar dat moet aangepast worden aan het nummer dat het icoon Cancel bij jou heeft. Het is natuurlijk mooi om Cancel te veranderen in Quit, Klaar of Einde.

Testen of het werkt en als je nu met je muis over 'main' gaat, hoor en zie je vanalles. Het aardige is dat ook zichtbaar wordt wat RISC OS voor je doet. Klik maar eens met verschillende knoppen op de verschillende iconen en zie wat er in het venster op verschillende plaatsen zichtbaar gemaakt wordt. Hou de linker of rechterknop een tijdje ingedrukt op een icoon en je ziet dat de waarde van de rechterknop dan 16 en die van de linker 64 wordt. Kennelijk is er verschil tussen kort klikken en lang ingedrukt houden.

Nu kan PROCmuis voor algemeen gebruik veranderd worden in:

```
DEF PROCmuis(x%,y%,knop%,venster%,icoon%)
  IF (venster%=hmain%) AND (knop%=4) AND (icoon%=11) THEN klaar%=TRUE
ENDPROC
```

### 6.3 invoer uit iconen

De gebruiker kan ook een icoon als invoerveld gebruiken. Dan moet de waarde die daarin staat, worden gelezen en in een variabele terecht komen. Met die variabele kan dan vanalles gedaan worden. Er zijn twee vragen: hoe weet het programma dat er invoer heeft plaatsgevonden en hoe wordt de invoer vanuit het icoon doorgegeven aan ons programma.

Een invoerreeks wordt door de gebruiker afgesloten met een Return en die kunnen we gebruiken om het proces te starten.

Er is een event op plaats 8 Key\_Pressed en die kunnen we afvangen met:

```
WHEN 8:PROctoets
```

Nu zorgt RISC OS dat er in het blok een hoeveelheid data wordt geplaatst en die moet geïnterpreteerd worden.

Het blok bevat:

blok+0	window handle
4	icon handle
8	x-offset of caret
12	y-offset of caret
16	caret hoogte
20	index van caret
24	character code van toets, die van Return is 13 of &0D.

de eerste twee en die laatste hebben we nodig, de overigen komen later aan de orde. De procedure-aanroep wordt nu vergelijkbaar met die van de muis:

```
WHEN 8:PROCToets(blok%!0,blok%!4,blok%!24)
```

En de uitwerking:

```
DEF PROCToets(venster%,icoon%,toets%)
  IF toets%=13 PROCinvoer(venster%,icoon%) ELSE SYS "Wimp_ProcessKey",toets%
ENDPROC
```

Het werkt nu als volgt: de gebruiker voert iets in en sluit dat af met Return. Die toetsaanslag wordt gesignaleerd door event 8 en data wordt in het blok geplaatst. Dat controleren we met PROCToets. Vervolgens wordt PROCinvoer gestart om na te gaan wat de gebruiker in dat icoon geplaatst heeft en kent dat toe aan een variabele.

ProcessKey

Voor dat we zien hoe dat moet, eerst even de SWI ProcessKey. Ons programma gedraagt zich in multitasking. Er zijn toetsaanslagen die voor het systeem gelden, denk aan F12 of Alt-Break. Die mogen door ons programma niet worden tegengehouden en worden met deze SWI doorgegeven. Daarom moet bij event 8, na het gedeelte waar de toetsaanslagen verwerkt worden, altijd deze SWI vermeld worden.

We zagen in de vorige paragraaf al dat met GetIconState de huidige inhoud van een icoon in het blok geplaatst wordt. Als de huidige inhoud dat is wat de gebruiker erin gezet heeft, dan staat die inhoud nu in het blok. En daaruit kan het, vanaf byte 28, gelezen en toegekend worden aan een variabele, in het voorbeeld A\$.

De constructie wordt als:

```
DEF PROCinvoer(venster%,icoon%)
  !blok%=venster%:blok%!4=icoon%
  SYS "Wimp_GetIconState",,blok%
  A$=$blok%!28
ENDPROC
```

Wij, de makers van ons programma, weten wat er in een bepaald icoon ingevuld moet worden. Of dat een tekst of een getalswaarde is. Nu wordt de inhoud altijd als string aan ons doorgegeven, maar die moet nog verder worden bewerkt als het een getal is.

VAL

Dat doen we met de Basic-opdracht VAL:  
waarde=VAL(A\$) of waarde%=VAL(A\$)

Zet de aanvullende programmaregels uit deze paragraaf in je programma, bewaar het en kijk of het werkt.

PROCinvoer kan ook met de muis gestart worden. Stel je voor dat er in een venster naam, adres en woonplaatsgegevens worden gevraagd. Het heeft niet zoveel zin om iedere ingevoerde regel na een Return meteen te laten verwerken. Je zou een OK-knopje kunnen maken. Als alles is ingevuld, klik je op OK. Maar dan komt de verwerking te liggen bij PROCmuis.

Maak nu, bij wijze van oefening, een programma met alles wat erbij hoort (dus ook Sprites, Help-file enzovoorts) waarbij de diameter van een cirkel als invoer wordt gevraagd en dat in twee iconen de omtrek en de oppervlakte laat zien. De formules zijn:

omtrek=PI\*diameter

oppervlakte=(PI/4)\*(diameter^2)      ( $\frac{1}{4}\pi D^2$ )

waarbij PI als Basic-opdracht kan worden gebruikt (RISC OS kent de waarde ervan).

Zorg voor een Quit-knop in het venster.

Hou er ook rekening mee dat de verschillende waarden als gebroken getallen gebruikt moeten kunnen worden en dat je daarvoor een variabele van het type real moet gebruiken (zonder %-teken).

Het is ook mogelijk met al het voorgaande een belastingprogramma te maken. Het is alleen wat meer werk, maar moeilijker is het niet.

En wie van muziekjes houdt, kan dit idee misschien gebruiken voor het maken van een muisgestuurd orgeltje.

Succes.

Aan het einde van dit hoofdstuk is !RunImage als volgt:

```
ON ERROR PROCerror:END
```

```
PROCbegin
PROChoofd
PROCeinde
END
```

```
DEF PROCbegin
  DIM blok% 255, wmain% xxx, winfo% yyy, naam1% 11, naam2% 11
  klaar%=FALSE:versie%=4.02*100:app$="MijnProg":filter%=&1C33
  SYS "Wimp_Initialise",versie%,&4B534154,app$ TO ,taak%
  SYS "Wimp_OpenTemplate",,"<MijnProg$Dir>.Templates"
  $naam1%="main":$naam2%="info"
  SYS "Wimp_LoadTemplate",,blok%,wmain%,wmain%+xxx,-1,naam1%,0
  SYS "Wimp_CreateWindow",,blok% TO hmain%
  SYS "Wimp_LoadTemplate",,blok%,winfo%,winfo%+yyy,-1,naam2%,0
  SYS "Wimp_CreateWindow",,blok% TO hinfo%
  SYS "Wimp_CloseTemplate"
  !blok%=hmain%:SYS "Wimp_GetWindowState",,blok%
  SYS "Wimp_OpenWindow",,blok%
  REM !blok%=hinfo%
  REM SYS "Wimp_OpenWindow",,blok%
ENDPROC
```

```

DEF PROChoofd
  WHILE klaar%=FALSE
    SYS "Wimp_Poll",filter%,blok% TO reason%
    CASE reason% OF
      WHEN 2:SYS "Wimp_OpenWindow",,blok%
      WHEN 3:SYS "Wimp_CloseWindow",,blok%:klaar%=TRUE
      WHEN 6:PROCmuis(blok%!0,blok%!4,blok%!8,blok%!12,blok%!16)
      WHEN 8:PROctoets(blok%!0,blok%!4,blok%!24)
      WHEN 17,18:IF blok%!16=0 THEN klaar%=TRUE
    ENDCASE
  ENDWHILE
ENDPROC

DEF PROCeinde
  SYS "Wimp_CloseDown",taak%,&4B534154
ENDPROC

DEF PROCerror
  !blok%=ERR
  $(blok%+4)=REPORT$+" (in regel "+STR$ ERL+" )"+CHR$0
  SYS "Wimp_ReportError",blok%,1,app$
ENDPROC

DEF PROCmuis(x%,y%,knop%,venster%,icoon%)
  IF venster%=hmain% AND knop%=4 AND icoon%=11 THEN klaar%=TRUE
ENDPROC

DEF PROCicontekst(naam%,icon%,text$)
  !blok%=naam%:blok%!4=icon%
  SYS "Wimp_GetIconState",,blok%
  $blok%!28=text$
  blok%!8=0:blok%!12=0
  SYS "Wimp_SetIconState",,blok%
ENDPROC

DEF PROctoets(venster%,icoon%,toets%)
  IF toets%=13 PROCinvoer(venster%,icoon%) ELSE SYS
"Wimp_ProcessKey",toets%
ENDPROC

DEF PROCinvoer(venster%,icoon%)
  !blok%=venster%:blok%!4=icoon%
  SYS "Wimp_GetIconState",,blok%
  A$=$blok%!28
ENDPROC

```

In de volgende les gaan we ons programma op de icoonbalk zetten en voegen we een menu toe.

In dit hoofdstuk zijn 2 paragrafen:

7.1 de icoonbalk 39

7.2 menu's 40

## 7.1 de icoonbalk

We kunnen ons programma direct laten starten als het wordt aangeklikt, zoals we het nu voor elkaar hebben, maar we kunnen er ook voor zorgen dat het eerst op de icoonbalk verschijnt zodat het van daaruit gestart kan worden. We moeten daar wel bewust voor kiezen, want er moet een reden voor zijn. We willen bijvoorbeeld het programma vast klaar kunnen zetten voor gebruik of het in de startprocedure, de boot, opnemen zodat we het niet uit de verste uithoek van de schijf hoeven op te halen. Maar het mag ook direct starten.

Omdat we de keuze willen hebben, komt hier het programmadeel.

We kijken eerst naar de gang van zaken zoals die er nu is en dan naar de startprocedure zoals die moet gaan worden. Zo kunnen we zichtbaar maken waar we de aanpassingen moeten doen.

In PROCbegin wordt de template aangeroepen, het venster gecreëerd en op het scherm gezet (met "Wimp\_GetWindowState" en "Wimp\_OpenWindow") en in PROC hoofd wordt het venster op het scherm gehouden met "Wimp\_OpenWindow". Het aanroepen en creëren blijven, maar in plaats van de eerste "Wimp\_GetWindowState" en "Wimp\_OpenWindow" moet 1 het icoon op de balk komen met "Wimp\_CreateIcon", 2 moet PROC muis signaleren dat daarop geklikt wordt en 3 moet dan "Wimp\_GetWindowState" en "Wimp\_OpenWindow" worden uitgevoerd.

Ten eerste vraagt "Wimp\_CreateIcon" de volgende waarden in het blok:

blok+0	handle voor icoon of -2 voor links of -1 voor rechts op de balk
4	minimum x-positie van icoon
8	minimum y-positie van icoon
12	maximum x-positie
16	maximum y-positie
20	icoon vlaggen
24	12 bytes icoondata

links of rechts op de balk

We willen ons icoon rechts op de balk, zoals gebruikelijk voor programma's (want apparaten komen links) en we geven dus -1.

Op plaatsen 4 en 8 moeten 0 worden gezet en op 12 en 16 komen 68. Dat zijn 68 OS-units die overeenkomen met 17 pixels. Dit zijn standaardwaarden en die handhaven we maar.

Op 20 plaatsen we voorlopig &3002. Later brengen we details aan.

Op 24 komt de naam van ons programma te staan: "!MijnProg"

Verder is het handig, later met menu's, om hiervoor een eigen handle te gebruiken: `ihandle%`.

Het wordt dus:

```

blok%!0=-1:blok%!4=0:blok%!8=0:blok%!12=68:
blok%!16=68:blok%!20=&3002:$(blok%+24)="!MijnProg"
SYS "Wimp_CreateIcon",,blok% TO ihandle%

```

Deze twee regels kunnen in het programma.

grootte van icoon

Ten tweede en ten derde moet PROCmuis worden uitgebreid met de volgende regel:  
 IF (venster%=-2) AND (knop%=4) !blok%=hmain%:SYS  
 "Wimp\_GetWindowState" , ,blok%:SYS "Wimp\_OpenWindow" , ,blok%

Ook die regel kan worden toegevoegd. Jouw editor breekt de regel waarschijnlijk wat ongelukkig af: bovenstaande programmaregel is maar een regel.

We moeten bij venster% niet de handle van het venster geven (hmain%), want dat venster staat nog niet op het scherm. Met -2 geven we aan dat het om een icoon op de icoonbalk gaat. Wordt daar met de linkermuisknop op geklikt, dan wordt hetzelfde gedaan wat er eerst in PROCbegin gebeurde.

Overigens is het hier ook mogelijk een menu te openen met de opdracht:

```
IF (venster%=-2) AND (knop%=2) PROCmenu
al wordt dat in de volgende paragraaf verder uitgewerkt.
```

## 7.2 menu's

middelste  
muisknop

Een menu start met een klik met de middelste muisknop op het icoon op de balk of in het venster van het programma. Dan zijn er twee mogelijkheden: de gebruiker maakt een keuze (die gesignaleerd en verder verwerkt moet worden) of de gebruiker klikt ergens anders waardoor dat menu weer verdwijnt.

menu-items

In een menu moeten verschillende dingen komen te staan: de naam van het programma, waarbij de naam een andere achtergrond heeft, dan Info en Quit en bij Info een pijltje dat naar rechts wijst. Als we met de muis daarlangs gaan of op Info klikken, moet het venster van template 'info' zichtbaar worden. Bij Quit moet het programma natuurlijk stoppen.

We laten PROCmuis signaleren dat er om een menu wordt gevraagd. Dan wordt PROCmenu geactiveerd. De gebruiker maakt een keuze. Die wordt uitgevoerd. Als er geen keuze wordt gemaakt, hoeven we niets te doen, omdat het verdwijnen van het menu door RISC OS wordt verzorgd.

In PROCmuis moet de combinatie vastgesteld worden van knop%=2 en venster%=-2 of venster%=hmain%. In beide gevallen moet PROCmenu aangeroepen worden:

```
IF (venster%=hmain%) AND (knop%=2) PROCmenu
```

```
IF (venster%=-2) AND (knop%=2) PROCmenu
```

Deze regels kunnen al worden toegevoegd aan het programma.

Wat moet er in PROCmenu?

De SWI is SYS "Wimp\_CreateMenu",R0,R1,R2,R3

R0 wordt niet gebruikt

R1 wordt een blok. We maken er een eigen blok van met de naam menublok%. Dat moet in PROCbegin bij de arrays worden aangemeld met een grootte. De grootte wordt bepaald door 28 bytes voor het uiterlijk en 24 bytes per item. Hier is grootte dus  $28+(2*24)=76$ . Het wordt dus DIM menublok% 76.

R2 bevat de x-positie van het menu

R3 bevat de y-positie van het menu.

We hebben dit in hoofdstuk 4.6 al gezien, bij wijze van voorbeeld hoe een blok gevuld moet worden. Kijk eventueel het desbetreffende deel nog even na om te zien hoe dat moest.

Voor het hele menu:

menublok%+ 0    menutitel  
12    voorgrondkleur van titel en frame (meestal zwart)  
13    achtergrondkleur van titel (meestal grijs)  
14    voorgrondkleur van menutekst (meestal zwart)  
15    achtergrondkleur van menutekst (meestal grijs of wit)  
16    breedte van de menu-items  
20    hoogte van ieder menu-item  
24    tussenruimte van menu-items  
28    opsomming van menu-items (24 bytes voor ieder item)

Per menu-item:

bytes 0..3    menu flags  
Bit    Betekenis  
0    een vinkje links van item  
1    onderbroken streep onder item  
2    item is beschrijfbaar  
3    toon bericht  
7    voor het laatste item in het menu als afsluiting

alle andere bits zijn 0

bytes 4..7    submenu pointer ( $\geq 8000$ ) of  
window handle (1-7FFF) of  
-1 indien niet gebruikt

bytes 8..11    menu icon flag  
bytes 12..23    menu icon data

De menutitel plaatsen we als volgt:

\$menublok%="MijnProg". Die naam zit al in de variabele app\$ en dus wordt het:  
\$menublok%=app\$

menutitel

De kleur van de voorgrond van de titel is zwart, waarde 7 (we handhaven de conventies van RISC OS):

menublok%?12=7

voorgndkleur

Achtergrondkleur wordt grijs met:

menublok%?13=2

achtergrondkleur

De kleur van de items wordt ook zwart:

menublok%?14=7

menu-item  
voorgnd

De achtergrondkleur is wit (0) of grijs (2)

menublok%?15=2

achtergrond

Je zou ook de vier bytes in een keer met een woord kunnen schrijven als (van achter naar voor):

menublok%!12=02070207

Op plaats 16 wordt de breedte van het menu aangegeven. Die is te bepalen, want daar geldt natuurlijk die van het breedste item. Maar dat moet worden uitgedrukt in OS-units en daarvoor wordt die breedte min 2 vermenigvuldigd met 16. De titel is het breedst (8 tekens-2) maal 16 is 96 units:

menublok%!16=96

breedte

hoogte Op 20 wordt de hoogte vermeld. 44 is precies goed al kun je ook eens andere waarden bekijken:  
menublok%!20=44

tussenruimte Ook de tussenruimte is kennelijk variabel en die geven we op 24 aan met 0.  
menublok%!24=0

Voor ieder menu-item moet er nu een stuk van 24 bytes beschreven worden en worden er vier aspecten gevraagd: menuvlaggen, een pointer, icoonvlaggen en data. Het volgende overzichtje maakt duidelijk waar voor drie items de zaken moeten worden geschreven:

menuvlaggen	menuvlaggen	28	52	76	100	124
pointer	pointer	32	56	80	104	128
icoonvlaggen	icoonvlaggen	36	60	84	108	132
data	data	40	64	88	112	136

De strekking is dat er bij ieder volgend item steeds 24 bijkomt. Wij hebben de eerste twee ervan nodig.

Het eerste item is Info. Daar komen geen vlaggen bij te staan en de bijbehorende waarde is dus 0. Het tweede item is Quit en omdat dat het laatste item is, moet bit 7 op 1 staan. De bijbehorende waarde is %10000000 of &80.

De pointer voor Info gaat de handle bevatten voor template 'Info' en die handle is hinfo%. De pointer voor Quit is -1.

De icoonvlaggen voor Info en Quit worden &7000021. Dat is een woord met vier bytes. Ze zeggen iets over ondermeer de kleuren van de tekst.

De data voor Info wordt "Info" en die voor Quit wordt "Quit".

Invullen maar:

```
menublok%!28=0:menublok%!32=hinfo%:menublok%!36=&7000021
$(menublok%+40)="Info"
menublok%!52=&80:menublok%!56=-1:menublok%!60=&7000021
$(menublok%+64)="Einde"
```

En nu kan "Wimp\_CreateMenu" worden aangeroepen. Het blok dat in R1 moet is menublok%.

In R2 en R3 komen de coördinaten van het menu te staan. De plaats van het menu is echter afhankelijk van de plaats van aanroep. Dat kan vanaf de icoonbalk, maar ook vanuit het venster. De coördinaten verschillen daardoor ook. We kunnen dat oplossen door de coördinaten mee te geven met de aanroep. We moeten PROCmenu dus uitbreiden. Bij die wordt aangeroepen vanaf de menublok met bijvoorbeeld !blok%-80,180 (verander deze waarden rustig en zie wat er gebeurt) en die van het venster met de cursorpositie. Die zijn bekend. PROCmuis heeft ze namelijk al gekregen in x% en y%.

```
IF (venster%=hmain%) AND (knop%=2) PROCmenu(x%,y%)
IF (venster%=-2) AND (knop%=2) PROCmenu(!blok%-80,180)
```

We gaan de procedure uitwerken:

```

DEF PROCmenu(x%,y%)
  $menublok%=app$:menublok%?12=7:menublok%?13=2
  menublok%?14=7:menublok%?15=0
  menublok%!16=96:menublok%!20=44:menublok%!24=0
  menublok%!28=0:menublok%!32=hinfo%
  menublok%!36=&7000021
  $(menublok%+40)="Info"
  menublok%!52=&80:menublok%!56=-1:menublok%!60=&7000021
  $(menublok%+64)="Einde"
  SYS "Wimp_CreateMenu",,menublok%,x%,y%
ENDPROC

```

Breng bovenstaand programmadeel aan, bewaar het geheel en test het. Als het werkt, zul je het menu zien en als je bij Info de pijl volgt, komt onze template 'Info' op het scherm. Dat doet RISC OS voor ons. Mooi hè?

Toch is het niet volmaakt. Telkens als het menu wordt aangeroepen, wordt het blok menublok% gevuld. En dat hoeft maar eenmalig, nu kost dat steeds extra tijd waardoor je programma trager wordt. Je kunt dat oplossen door het vullen van menublok% in de initialisatie te laten plaatsvinden, maar dan wordt dat zo vol. Vanuit PROCbegin kun je natuurlijk wel PROCvulmenu starten. Waar moet die komen te staan?

In de eerste alinea van deze paragraaf hebben we al vastgesteld dat de menukeuze ook verwerkt moet worden. Want met bovenstaande komt het menu wel op het scherm, maar reageert nog niet voldoende. We hebben event 9. Die ontvangt de menu-acties en kan die doorgeven. Dat doen we met een procedure:

verwerking van  
menukeuze

```
WHEN 9:PROCmenukeuze
```

```

DEF PROCmenukeuze
  REM verwerk de keuze van de gebruiker
ENDPROC

```

De keuze van de gebruiker is door RISC OS in blok% geplaatst. We gaan nog niet op allerlei details in, dat komt in het vervolg van de cursus. Maar we kennen de inhoud van het blok toe aan een variabele, we noemen die maar keuze%:

```
keuze%=!blok%
```

Het bovenste menu-item is 0, de volgende is 1 etc. We kunnen testen als:

```

IF keuze%=0
IF keuze%=1

```

Bij de laatste is het wel duidelijk, dan moet het programma stoppen door klaar%=TRUE.

Maar keuze%=0 vraagt wat extra aandacht. We zagen al dat template 'Info' spontaan verschijnt als we de muispijl langs de > bewegen. Maar toch moet het item Info ook aangeklikt kunnen worden. We doen dat met de regels die we in PROCbegin met een REM inactief hebben gemaakt:

```

IF keuze%=0 !blok%=hinfo%:SYS
"Wimp_GetWindowState",,blok%:SYS
"Wimp_OpenWindow",,blok%

```

Invoeren en testen maar.

Tot slot. We hebben nu enkele goede manieren om ons programma te eindigen. Die van event 17,18 zijn nu niet meer nodig. Ze mogen eruit, maar dat doen we niet.

Hiermee kan ons programma altijd via Tasks (achter de Switcher) worden beëindigd.  
 Eventuele REM-regels in PROCbegin mogen wel verwijderd worden. Wat er overblijft is een goed werkend programma.  
 Beschouw dit geheel maar als de basis voor verschillende programma's.

```

REM > !RunImage

ON ERROR PROCerror:END

PROCbegin
PROChoofd
PROCeinde

END

DEF PROCbegin
  DIM blok% 450, naam1% 11, wmain% 390, naam2% 11, winfo% 450, menublok%
  80
  klaar%=FALSE:versie%=402:app$="MijnProg":filter%=&E1833
  SYS "Wimp_Initialise",versie%,&4B534154,app$ TO ,taak%
  SYS "Wimp_OpenTemplate",,"<MijnProg$Dir>.Templates"
  $naam1%="main":$naam2%="info"
  SYS "Wimp_LoadTemplate",,blok%,wmain%,wmain%+390,-1,naam1%,0
  SYS "Wimp_CreateWindow",,blok% TO hmain%
  SYS "Wimp_LoadTemplate",,blok%,winfo%,winfo%+450,-1,naam2%,0
  SYS "Wimp_CreateWindow",,blok% TO hinfo%
  SYS "Wimp_CloseTemplate"
  blok%!0=-1:blok%!4=0:blok%!8=0:blok%!12=100
  blok%!16=100:blok%!20=&3002:$(blok%+24)="!MijnProg"
  SYS "Wimp_CreateIcon",,blok% TO ihandle%
ENDPROC

DEF PROChoofd
  WHILE klaar%=FALSE
    SYS "Wimp_Poll",filter%,blok% TO reason%
    CASE reason% OF
      WHEN 2:SYS "Wimp_OpenWindow",,blok%
      WHEN 3:SYS "Wimp_CloseWindow",,blok%:klaar%=TRUE
      WHEN 6:PROCmuis(blok%!0,blok%!4,blok%!8,blok%!12,blok%!16)
      WHEN 8:PROCtoets(blok%!0,blok%!4,blok%!24)
      WHEN 9:PROCmenukeuze
    ENDCASE
  ENDWHILE
ENDPROC

DEF PROCeinde
  SYS "Wimp_CloseDown",taak%,&4B534154
ENDPROC

DEF PROCerror
  !blok%=ERR
  $(blok%+4)=REPORT$+" (in regel "+STR$ ERL+)"+"CHR$0
  SYS "Wimp_ReportError",blok%,1,app$
ENDPROC

```

```

DEF PROCmuis(x%,y%,knop%,venster%,icoon%)
  IF venster%=hmain% AND knop%=4 AND icoon%=6 THEN klaar%=TRUE
  IF venster%=hmain% AND knop%=2 PROCmenu(x%,y%)
  IF venster%=-2 AND knop%=4 !blok%=hmain%:SYS
"Wimp_GetWindowState",,blok%:SYS "Wimp_OpenWindow",,blok%
  IF venster%=-2 AND knop%=2 PROCmenu(!blok%-80,180)
ENDPROC

DEF PROCicontekst(naam%,icon%,text$)
  !blok%=naam%:blok%!4=icon%
  SYS "Wimp_GetIconState",,blok%
  $blok%!28=text$
  blok%!8=0:blok%!12=0
  SYS "Wimp_SetIconState",,blok%
ENDPROC

DEF PROctoets(venster%,icoon%,toets%)
  IF toets%=13 PROCinvoer(venster%,icoon%) ELSE SYS
"Wimp_ProcessKey",toets%
ENDPROC

DEF PROCinvoer(venster%,icoon%)
  !blok%=venster%:blok%!4=icoon%
  SYS "Wimp_GetIconState",,blok%
  A$=$blok%!28
  REM voorbeeld van verwerking
  REM diam=VAL(A$):omtrek=PI*diam:opp=(PI/4)*(diam*diam)
  REM PROCicontekst(hmain%,3,STR$(omtrek))
  REM PROCicontekst(hmain%,4,STR$(opp))
ENDPROC

DEF PROCmenu(x%,y%)
  $menublok%=app$:menublok%?12=7:menublok%?13=2:menublok%?14=7:menublok%
?15=0
  menublok%!16=96:menublok%!20=44:menublok%!24=0
  menublok%!28=0:menublok%!32=hinfo%:menublok%!36=&7000021:$(menublok%+4
0)="Info"
  menublok%!52=&80:menublok%!56=-1:menublok%!60=&7000021:$(menublok%+64)
="Einde"
  SYS "Wimp_CreateMenu",,menublok%,x%,y%
ENDPROC

DEF PROCmenukeuze
  keuze%=!blok%
  IF keuze%=0 !blok%=hinfo%:SYS "Wimp_GetWindowState",,blok%:SYS
"Wimp_OpenWindow",,blok%
  IF keuze%=1 klaar%=TRUE
ENDPROC

```



In dit hoofdstuk staan de volgende paragrafen:

8.1 de syntactische fouten	47
8.2 de run-time errors	48
8.3 de logische fouten	49
8.4 systematisch testen	49
8.5 wat hoef je niet te testen?	50
8.6 het afhandelen van fouten	50
8.7 meldingen	51
8.8 communicatieve fouten	51

Voordat we aandacht besteden aan het opsporen en verhelpen van fouten, zullen we eerst moeten vaststellen welke typen fouten er zijn. Dat maakt gericht zoeken mogelijk. Er zijn twee hoofdgroepen: de technische en de communicatiefouten. De technische bestaan uit programmeerfouten als syntactische fouten of syntax errors, de run-time errors en de logische fouten. De communicatieve ontsparingen hebben betrekking op het verschil tussen dat wat de gebruiker ziet op het scherm en dat wat jij ermee bedoeld had. Het gaat hierbij om aspecten van software-ergonomie.

typen fouten

syntactische fouten

run-time errors

logische fouten

software-

ergonomie

Door zorgvuldige voorbereiding van je programma, vooraf en achteraf testen en zorgvuldigheid beperk je de kans op fouten enorm.

Die zorgvuldige voorbereiding houdt in dat je zorgt dat je het programma niet direct vanuit je gedachten invoert in het systeem. Zet het op papier. Teken het venster en zet ook je programma(concept) op papier.

Dat concept mag natuurlijk best in dagelijkse bewoordingen opgesteld worden als:

Open bestand

Schrijf variabelen

Sluit bestand

of als

Als venster=main en icoon=12 en knop=4 dan berekening (resultaat => icoon 15)

Met lijnen en pijlen kun je aangeven waar de invoer naar toe moet en in welke iconen de uitvoer komt te staan. Je kunt iconen alvast nummeren en die nummers kun je in het concept aangeven. Streep door, kruis aan, omcirkel, verplaats, hernoem, schaaft en beitel tot je het idee volkomen duidelijk voor ogen staat. Je kunt pas blokken vullen met data als je duidelijk weet wat die data moet zijn. Programmeren is vooral papierwerk en het coderen (omzetten in BASIC-opdrachten en invoeren) gebeurt pas in de voorlaatste fase. De laatste is testen.

testen

### 8.1 de syntactische fouten

Syntax heeft betrekking op de zinsbouw. De zin: 'Kan programmeren beetje ik' is syntactisch onjuist, hoewel wij flexibel genoeg zijn de strekking ervan te begrijpen. De computer mist het vermogen tot flexibiliteit of improvisatie en is niet in staat er chocola van te maken, hetgeen ons dwingt tot zorgvuldig formuleren. Meestal hebben syntactische fouten betrekking op type- of spelfouten. Daardoor wordt een variabele, een SWI of opdracht niet of onjuist gebruikt. Aangezien de computer er niets mee kan, stopt de verwerking hier en is een foutmelding nog het enige dat het programma kan verstreken, als dat nog lukt. Soms is het resultaat dat het systeem hangt en is er nog slechts een Alt-Break mogelijk om het programma te stoppen. De variabele handle% mag dus niet als handel% worden geschreven: de foutmelding is

Alt-Break

'Unknown or missing variable'. De opdracht WHILE mag niet als WHLE worden geschreven, want dat leidt tot de melding 'Mistake' en SYS "Wimp\_initialize" leidt tot 'SWI name not known'.

Syntax errors leveren duidelijke foutmeldingen maar tijdens het testen moeten per se alle procedures minstens een keer verwerkt worden.

visuele controle

Een mogelijkheid om de kans op syntactische fouten te verkleinen is het programma zeker een keer grondig visueel te controleren. Systematisch. Bovenaan beginnen en regel voor regel zorgvuldig lezen. Aangezien het lezen vanaf het scherm toch lastig is, zou je het programma ook af kunnen drukken, want het lezen vanaf papier is minder vermoeiend. En je kunt er zo fijn aantekeningen op maken.

Ten tweede is het bijzonder handig een lijstje aan te leggen van gebruikte variabelen. Daarin staat natuurlijk de correcte schrijfwijze, zodat je je daarmee niet hoeft te vergissen.

## 8.2 run-time errors

Ook bij dit fouttype kan er sprake zijn van een typefout, maar dan met een andere uitwerking. Een komma maakt al wat uit. De SWI SYS "Wimp\_Initialize",R0,R1,R2 TO R0,R1 kan achter TO twee variabelen vullen: versie% en handle%. Als je alleen handle% gebruikt, zul je in plaats van versie% een komma moeten plaatsen omdat de waarde die in R0 staat anders in handle% geplaatst wordt. Hiermee komt een goede waarde in de verkeerde variabele terecht.

Bij run-time errors bestaat de mogelijkheid dat het programma niet verder kan, maar er zijn situaties waarin het programma niet stopt, maar met verkeerde waarden verdergaat.

Waar hebben run-time errors betrekking op?

Denk aan variabelen die geen waarde hebben gekregen en waar toch iets aan verwerkt wordt.

Of aan variabelen die met verkeerde waarden worden gevuld. Een variabele van het type string (var\$) die met een getalswaarde wordt gevuld of omgekeerd, een getalsvariabele (var%) die met een tekst wordt gevuld.

Denk aan een blok of array dat niet groot genoeg gedefinieerd is.

Of een opdracht die te vroeg in het programma gegeven wordt, zodat de bijbehorende data nog niet compleet is.

Of een opdracht die te laat gegeven wordt, waardoor de inhoud van een blok alweer veranderd is.

Of een test die onzorgvuldig is uitgewerkt. IF A=PI AND B=4 kan leiden tot de bewerking PI AND B. Daarom worden de leden van de vergelijking tussen haakjes gezet: IF (A=PI) AND (B=4).

Het nadeel van dit fouttype is dat de melding soms minder duidelijk is. 'Internal error: abort on data transfer' of 'There is not enough memory to create this window' hebben betrekking op arrays die te klein gedefinieerd zijn, maar de melding 'Unable to bind fonts' heeft daar ook mee te maken. Soms treedt de fout op een heel andere plaats aan het licht dan waar hij ontstaan is.

Ook bij dit type geldt dat je ieder deel van het programma tijdens het testen minstens een keer moet laten verwerken.

### 8.3 de logische fouten

Deze vallen eigenlijk niet in de categorie programmeerfouten. Sterker nog, het programma kan volledig foutvrij zijn en toch een logische fout bevatten. Dit type heeft te maken met de wijze waarop het programmeerprobleem is opgelost. In het dagelijkse leven zouden we ze denkfouten noemen, de verkeerde conclusie: het algoritme, het oplossingsmodel, is verkeerd uitgewerkt. Ze zijn soms bijzonder moeilijk te vinden, simpelweg omdat het kan zijn dat je er vanuit gaat dat jouw oplossing de juiste is. De oppervlakte van een cirkel wordt berekend met eenvierde-pi-D-kwadraat oftewel  $0,25 \times 3,14 \times \text{diameter}$  in het kwadraat. Als je er van overtuigd bent dat de formule met de straal moet worden berekend, maak je onbedoeld een logische fout.

Er is wel een remedie voor. Controleer de oplossing handmatig. Voer zelf de berekeningen uit, zoek de juiste formules op, verifieer de gekozen oplossing op papier met verschillende waarden en vraag je af of er nog een andere oplossingsmogelijkheid bestaat. En realiseer je dat de computer alleen die problemen kan oplossen, die je zelf al opgelost hebt.

### 8.4 systematisch testen

Het uitgangspunt moet zijn dat je programma uiteindelijk nagenoeg foutvrij is. Als je dat idee zou loslaten, kom je op het hellende vlak waarop Windows zich ook bevindt.

Zet een testprocedure op. Druk dus niet zomaar op een knopje of voer niet lukraak wat in om te zien of het werkt, maar maak een plan en benoem daarin alles dat getest moet worden. Hoe groot mag eigenlijk het bedrag zijn dat je invoert? Maakt dat wat uit? Kun je een begrenzing in je programma aanbrengen zodat die invoer geen probleem zal vormen? Mogen er ook negatieve getallen in of is er op een andere manier iets te bedenken dat de gebruiker anders zou doen dan jij je voorstelt. Werkt je programma ook met grote bestanden? Negatieve getallen, gebroken getallen in plaats van alleen maar integere?

testprocedure

Jij hebt de functionaliteit van je programma bedacht. Jij weet wat er kan en daar kun je dus een lijstje van maken. En alle items van dat lijstje kunnen achtereenvolgens getest worden. Als je een nieuw element wilt toevoegen aan je programma, zou je dat dan ook vooraf kunnen testen? Ontdaan van de rest van het programma zodat er niets is dat de werking van het nieuwe programmadeel verstoort of beïnvloedt. Zodat je het pas inbouwt als het geen fouten meer bevat. Vergeet niet het geheel te testen.

Ook het gebruik van REM is aan te bevelen. Zet REM in of voor iedere regel die tijdelijk uitgeschakeld moet worden en die regels doen niet meer mee. Zo kun je procedures uitschakelen en wat directer nagaan of een bepaald deel doet wat jij wilt.

Het programma !Reporter kan je de inhoud van allerlei variabelen laten zien. Zet in jouw programma op verschillende plekken de opdracht \*Report var1 var2 var3 waarbij var1 tot en met 3 de naam van een variabele voorstelt die je wilt bekijken. Run !Reporter en run jouw programma. In een apart venstertje worden de namen en de waarden van de variabelen zichtbaar. Voor meer details is het raadzaam in de help-file van !Reporter te kijken.

Test je programma niet alleen op jouw systeem, met jouw RISC OS-versie en hardware, maar ook op dat van anderen. Noteer je bevindingen in de !Help-file. Je kunt ook anderen vragen om je programma te testen.

In het venster van de TaskManager (achter de switcher) kun je zien of je programma binnen de aangegeven geheugengrenzen blijft. Controleer dat ook eens.

Met de utility 'TaskUsage' kun je nagaan of je programma niet al teveel beslag legt op de processortijd. Is dat wel het geval (twintig procent of meer terwijl het programma in rust is) dan zou je nog eens naar het filter voor "Wimp\_Poll" kunnen kijken.

En dan is het mogelijk om een procedure in je programma op te nemen dat (fout)meldingen doorgeeft; we hebben er al een gezien in hoofdstuk 4. Met een beetje doordachte opzet kun je zorgen dat een duidelijke melding plus regelnummer zichtbaar wordt. Gedurende de ontwikkel- en testfase werkt dat prima al komt er een moment, als je het programma vrijgeeft, waarop die routine vervangen moet worden door een gebruikersvriendelijkere. Daarin horen geen systeemmeldingen of regelnummers.

```
DEF PROCerror
  !blok%=ERR
  $(blok%+4)=REPORT$+" (in regel "+STR$ ERL+)" +CHR$0
  SYS "Wimp_ReportError",blok%,1,app$
ENDPROC
```

Als door bovenstaande procedure een venster op het scherm verschijnt met de melding dat er een fout in regel 85 zit, hoe vind je dan regel 85? Zet de cursor ergens in de programmatekst en druk op menu. Via 'Edit' kun je naar Goto en daar vraagt het programma om een regelnummer. Voer 85 in en de cursor springt naar de desbetreffende regel.

## 7.5 wat hoeft je niet te testen?

Het klinkt wat merkwaardig, maar je hoeft niet te testen of jouw programma ook op oudere systemen draait. Dat is namelijk vooraf controleerbaar, zelfs op papier. Je zou daarvoor overigens de documentatie van voorgaande RISC OS-versies moeten hebben. Het volgende beeld kan dan ontstaan:

	RO2	RO3	RO4
SWI A	A	A	A
		B	B
	C	C	C
			D

Een programma dat SWI A gebruikt, kan op iedere RO-versie worden toegepast en als SWI D gebruikt wordt, draait het programma alleen op RO4.

Het spreekt, hoop ik, voor zich dat je wel zult moeten testen als je die documentatie niet hebt.

## 7.6 het afhandelen van fouten

We moeten wel een uitgangspunt nemen. Moet je programma stoppen bij een fout, of moet het juist doorgaan waarbij de gevolgen van de fout zoveel mogelijk opgevangen worden. Het eerste is natuurlijk het eenvoudigst: ON ERROR PROCerror:END De gebruiker krijgt nog een melding van het probleem dat opgetreden is en dan stopt het programma ermee. Maar misschien kan de gebruiker het probleem oplossen en zou het programma verder moeten. Misschien kan de programmeur een alternatief bedenken en toepassen, zodat het programma verder kan.

## 8.7 meldingen

We kunnen het programmadeel uit de vorige paragraaf ook gebruiken voor het doorgeven van meldingen. Als icoon 15 gebruikt wordt voor invoer, dan kunnen we testen of die invoer niet groter of kleiner is dan een bepaalde waarde. Als dat zo is, moet er melding van worden gemaakt en moet de gebruiker een nieuwe waarde invoeren.

We gebruiken:

```
SYS "Wimp_ReportError" ,blok%,vlag%,app$ TO ,errklik%
```

Waarbij het woord vlag% de volgende bitwaarden kan bevatten:

bit	0	toon een Ok-icoon
	1	toon een Cancel-icoon
	2	toont ze beide, maar benadruk Cancel
	3	toon niet: "Press space or click mouse to continue"
	4	begin niet met tekst: "Error from "
	5	gaat meteen verder (klik%=0) en het venster blijft open
	6	bootst klik% na afhankelijk van 0 en 1
	7	geen biep
	8	toegang tot extra
	9-11	extra (besteden we later aandacht aan)
	12-31	gereserveerd.

We gebruiken de bits 0, 3, 4 en 7. vlag% wordt dan  $0+4+8+64=76$ .

De test wordt dan:

```
IF invoer%>1000 PROCmelding("Invoer is te groot (max. 1000)",76) ELSE IF invoer%<=0 PROCmelding("Invoer is te klein (min. 1)",76)
```

PROCmelding(bericht\$) zou er dan zo uit kunnen zien:

```
DEF PROCmelding(bericht$,vlag%)  
$(blok%+4)=bericht$+CHR$(0)  
SYS "Wimp_ReportError" ,blok%,vlag%,app$ TO ,errklik%  
ENDPROC
```

Errklik% bevat het antwoord van de gebruiker: 0 is geen klik, 1 is geklikt op Ok, 2 is geklikt op Cancel en 3 tot 5 geklikt op extra knoppen die door de gebruiker zijn gemaakt. Het is aan de programmeur met die klikken iets te doen.

## 8.8 communicatieve fouten

Een programmeur heeft een half jaar hard gewerkt aan een applicatie. Na de invoering ervan blijkt het programma nog zoveel fouten te bevatten dat het onoverbrugbare problemen oplevert. Dat programma wordt al snel niet meer gebruikt.

Een andere programmeur heeft ook maandenlang gewerkt aan een toepassing en ook daarvan blijkt, na invoering, dat het vrijwel nergens gebruikt wordt. Onderzoek naar de oorzaak maakt duidelijk dat de gebruikersvriendelijkheid erg tegenvalt: gebruikers kunnen er niet mee overweg.

Hoewel de oorzaken verschillen, is het resultaat hetzelfde: de inspanningen van de programmeurs hebben niet geleid tot goede, bruikbare en betrouwbare programma's en de conclusie moet zijn dat de aard van de fouten er niet zoveel toe doet.

Programmeerfouten zijn door zorgvuldig testen nog wel op te sporen en te verbeteren. Bij communicatie tussen gebruiker en programma is dat een stuk moeilijker. De gebruikersinterface verdient dus apart aandacht, het liefst al in een vroeg stadium.

Waarom. Als je een programma hebt geschreven en getest, breng je het uit. Je stelt het ter beschikking. Als dan blijkt dat de gebruikersvriendelijkheid slecht uitvalt, moet je het hele programma herzien. Vensters moeten worden gewijzigd, misschien gesplitst, routines moeten veranderen, berichten ook. Je kunt dat voorkomen door het maken van een prototype. Daarbij komt het er op neer dat je eerst de vensters maakt, met knopjes om naar andere vensters te gaan, in- en uitvoervelden, menu's, berichten, maar zonder functionaliteit. Het knopje voor het afdrukken is er wel, maar het afdrukken zelf is nog niet aangebracht. Je kunt wel wat invoeren, maar de invoer wordt niet verwerkt, er zijn geen resultaten van bewerkingen zichtbaar en menukeuzes hebben nog geen effect.

Nu laat je het programma testen op gebruikersvriendelijkheid. Als het nodig is, breng je wijzigingen aan en dan pas voeg je de functionaliteit toe. Afdrukken, laden, bewaren en alle andere mogelijkheden die via knoppen of het menu toegepast kunnen worden.

Wat kan er in de communicatie misgaan?

Hieronder volgen aanbevelingen om het contact tussen programma en gebruiker te verbeteren. Ze hebben betrekking op verschillende aspecten.

leerbaarheid

leerbaarheid

Zorg dat het programma gemakkelijk te leren en te gebruiken is. Handhaaf de RISC OS-standaard zoveel mogelijk.

Zorg voor overzichtelijkheid van opties en iconen.

Zorg dat iconen duidelijk omschreven taken uitvoeren.

Splits complexe taken in series eenvoudiger taken.

Zorg voor een logische en duidelijke structuur zodat gebruikers niet teveel details of teveel stappen hoeven te onthouden.

invoer

invoer

De volgorde van invoervelden komt overeen met de volgorde van de items op het papieren invulformulier.

Laat gebruikers geen lange woorden intoetsen want dat vergroot de kans op fouten.

Het verplaatsen van de caret in invoervelden moet met muis en toets kunnen.

De maximale lengte van de invoer is te zien aan de lengte van een invoerveld.

Volg de leesrichting bij het invullen. Eerst van links naar rechts, dan van boven naar beneden.

Groepeer velden naar onderwerp.

meldingen

meldingen

Meldingen moeten informatief en feitelijk zijn en niet bestraffend, cryptisch of humoristisch.

Gebruik in vensters of berichten geen afkortingen.

Geef bij meldingen suggesties om het probleem op te lossen. Doe dat eenduidig en niet als: 'Golfapparaataansturing niet gevonden of geïnstalleerd'.

Eenvoudig en direct taalgebruik in berichten. Geen bijzinnen, geen ontkenningen in vragende zinnen. Niet meer dan een voorwaarde in een zin. Gebruik de actieve vorm.

Hoofd- en kleine letters zoals in normaal taalgebruik.

menu's

Zet in menu's de meest gemaakte keuzes bovenin.

Niet teveel keuzes in een menu. Men beschouwt zeven als het maximum.

Geef vervolgmenu's een titel die overeenkomt met de naam uit het vorige menu.

Als met de rechtermuisknop in een menu wordt gekozen, moet het menu op het scherm blijven staan.

Groep items die bij elkaar horen. Scheid ze eventueel met een stippellijn.

Tekst in menu's schrijven we links uitgevuld, eerste letter kapitaal, sneltoetsen rechts. Alle tekst in systeemfont.

menu's

taalgebruik

Niet meer dan een voorwaarde in een vraag: Wilt u dit bestand afdrukken en het programma stoppen? Maar: Afdrukken (Ja, Nee) en daarna Stoppen (Ja, Nee)

Gebruik geen jargon in meldingen of dialoogschermen. Ook geen cryptische boodschappen als: 'Adress exception at &0004A23'.

Stel vragen die maar op een manier kunnen worden uitgelegd.

Vermijd woorden als fataal, afgebroken of vernietigd, ze klinken verontrustend.

Wees beleefd en spreek de gebruiker met u aan. Geef een verontschuldiging als het programma vastloopt.

taalgebruik

knoppen, toetsen, functietoetsen

Zorg voor sneltoetsen (zoals Shift D) zodat gebruikers geen hele menusequenties hoeven te doorlopen.

Gebruik de standaardfuncties van functietoetsen:

F1 Help

F2 Laad document

Shift F2 Voeg document in

CTRL F2 Sluit venster

F3 Bewaar document

F4 Zoek/vervang

F5 Ga naar

F6 Sorteer

F8 Undo

F9 Redo

F12 \*-commando's

Shift F12 Icoonbalk op voorgrond

CTRL F12 Open taakvenster

CTRL Shift F12 sluit af

knoppen, toetsen,  
functietoetsen

Om van het ene invoericoon naar het andere te springen, kunnen we de cursorbesturingstoetsen of de Tab gebruiken. Een druk op de Returntoets om de invoer af te sluiten, zorgt dat de cursor of caret naar het volgende invoericoon gaat.

beeldscherm en kleuren

Gebruik blikvangers met mate.

Hou het beeld zo rustig mogelijk, zonder knippen of veel kleuren.

Handhaaf de standaardkleuren van vensterattributen zoveel mogelijk. Hooguit kan de achtergrond van het werkveld in een andere tint. De randen, balken en vensterknoppen behouden hun standaardkleur.

Hou rekening met verschillende beeldschermformaten.

Zorg voor voldoende contrast tussen tekst en achtergrond.

beeldscherm en  
kleuren

vensters

Als het werkgebied van een venster (workarea) groter is dan het zichtbare gebied (visible area) maak dan duidelijk dat het venster meer keuzemogelijkheden of invoervelden kent dan alleen de zichtbare.

Probeer alle knoppen en velden in een venster te zetten, maar maak zo snel mogelijk een splitsing in vensters als de hoeveelheid te groot wordt.

vensters

hulp

hulp

Hou rekening met ervaren en minder ervaren gebruikers; help iedere categorie op eigen niveau.

veiligheid

veiligheid

Maak het gebruikers mogelijk om fouten te herstellen. Denk aan Undo en Redo.

Vraag om bevestiging als een actie onomkeerbaar is. (Het wissen van bestanden bijvoorbeeld)

Zorg, hoe dan ook, dat bij programmaproblemen geen werk van de gebruiker verloren gaat. Dat kan door je programma te laten zorgen dat iedere tien minuten het invoerwerk gesaved wordt.

Hier eindigt hoofdstuk 8 en daarmee deel 1.