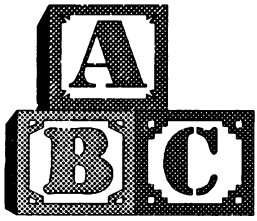
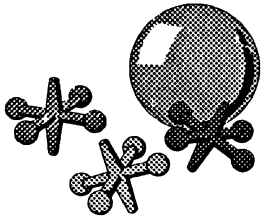

AMIGA
COMAL



TUTORIAL



REFERENCE

COMPILER

1. AmigaCOMAL tutorial.

1.1 Getting started.

After you have started AmigaCOMAL you will be presented with two windows. One large window and one smaller which is placed in the foreground covering a part of the large back ground window. The small window is active when you initially start AmigaCOMAL. This window is used for entering programs and commands to AmigaComal.

Let's start by writing a small program.

Type the following instruction:

```
10 PRINT "Good day!"
```

and remember always to end each line by pressing <RETURN>.

Now type:

```
RUN
```

remember <RETURN>.

You will notice if you are quick, that the small work window is placed in the back ground, and on the large window the text *Good day!* appears. Immediately afterwards the smaller programming window returns to the foreground again.

Everything happens so quickly, that you hardly notice that anything has happened, just a slight flickering of the terminal screen. You will however notice that the text *Good day!* appears in the large window. Type RUN again <return> so that you can see the window change and see your message. (In stead of typing RUN <return> you could press function key F5 to get the same effect).

Let's try to get our message *Good day!* placed in the center of the view screen. Move by using the arrow keys (or your mouse if you have one) the cursor so

that it is placed on the quote mark directly before the word *Good day!*. (click the mouse !!) Now type:

```
AT 10,30:
```

so that the new line looks like this:

```
10 PRINT AT 10,30: "Good day!"
```

Notice that the word "Good day" is automatically moved to the right as you type when you are finished typing press <RETURN>, type RUN <RETURN> (or use F5)

Nothing happened - or did there?

Let's just look at the line where we added, *AT 10,30:* This line say's that printing is to start on line 10 - position 30 in the window. Line 10 position 30 is however not visible as it is under our programming window. You can see this by pushing F5 a few times. This is a rather tedious way of reading information, so I think we should add a line to the program.

type :

```
20 WAIT
```

(remember <RETURN>).

Now you can see the text in the window. By using the instruction *WAIT* you have told AmigaCOMAL to stop and wait until another key is entered.

OK! push any key to see what happens.

You have now produced your first two line program. Congratulations, To see the entire program type:

```
LIST
```

(or you can use F1).

Now type:

```
NEW
```

(remember <RETURN>).

NEW is a command that prepares the Amiga for work, by deleting all current lines in memory. AmigaCOMAL ask's if you are sure? This time you can answer yes without worrying.

Now we will make a little larger program to show a few of the facilities in AmigaComal.

Type :

```
AUTO
```

(or press F2). In the programming window the number 0010 appears and the cursor is positioned directly after the number. Type :

```
page // Erase Window
```

and press <RETURN>. AmigaComal automatically writes the new line number 0020 and position the cursor ready for the next line.

Continue typing the following program in.

```
0010 page // Erase Window
0020
0030 INPUT AT 10,5: "Please enter your name: ": name$
0040 page
0050 PRINT AT 10,20: "Good day ",name$
0060 PRINT AT 12,20: "Welcome to AmigaCOMAL"
0070
0080 WAIT
```

When you have typed your program in, AMIGACOMAL automatically produces line 0090. To stop the automatic line numbering press the two key's at the same time <left Amiga> + <S>.

This key sequence <left Amiga> + <S> is used every time we wish to return to normal input in the programming window. This key combination can also be used to stop any program that is running (executing). This sequence is a 'short cut' to STOP the RUN instruction.

You can now run your program by typing RUN <RETURN> or use F5. Your program starts by erasing the window and the ask's you to type in your name. When you have done so and pressed <RETURN> You should find your

greeting in the middle of the screen. The *WAIT* Instruction as described before gives you time to enjoy the sight.

When you have seen enough press any key.

1.2 Graphic's.

The strength of the AmigaCOMAL is the fact that the language is interactive, and can therefore help you line for line with irritating small errors. You are able to RUN single lines of a program or entire procedure as commands. In the next example we will use a few of AmigaCOMALs graphic commands.

Before we can use the graphic capabilities of AmigaCOMAL we will load one of the standard packages, *UniGraphics* or *Turtle*. Let's choose the latter. To load the package you type:

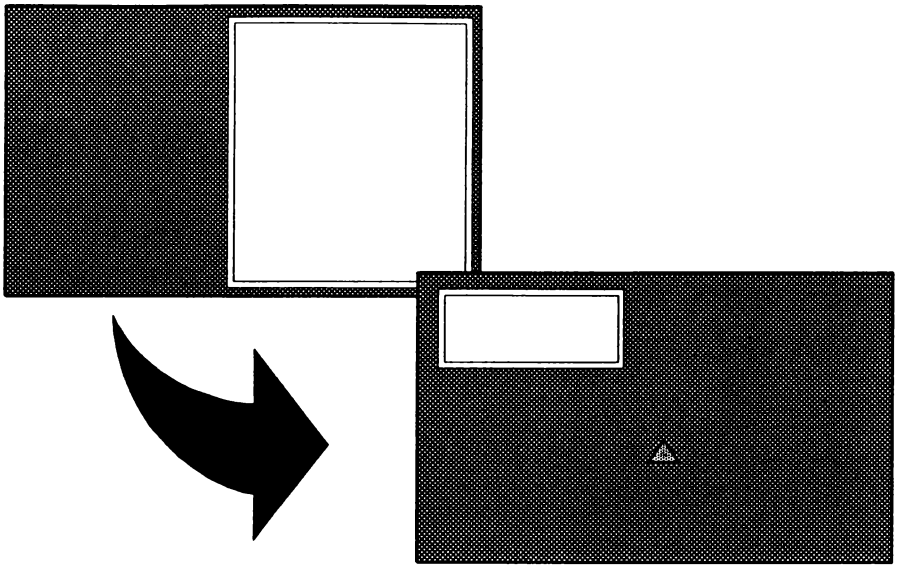
```
USE Turtle
```

AmigaCOMAL will then load the Turtle-package and make sure that all the necessary routines are available. In order to use the facilities it is necessary to open a window for the turtle to work in.... This is done by typing :.

```
graphicscreen(0)
```

and the usual <RETURN>.

The graphics window is not moved to the foreground and a little triangle is shown in the middle of the window. The triangle represent the turtle (thus the name *Turtle*).
The routines in the turtle package can now be used to direct the turtle around the screen any way you like. Re-organize the Amiga Window so that it look's like the screen below. If you have any problem's refer to the Amiga reference manual.



We are now ready to use a few of the turtle routines.

First we move the turtle forward, by typing:

```
fd(70)
```

(*fd* is the short form for *forward*). Now you want to turn 90 degrees to the right:

```
rt(90)
```

(*rt* is the short form for *right turn*). If we repeat these two instructions three times (the easy way is to move the cursor up two line and press <return> twice.)

As you can see we have drawn a square, but this can be done easier. Erase the screen so that we can continue - type :

```
cs
```

(this means *clear screen*). As you see the turtle has disappeared, but not really, it is actually still there. type:

```
LOOP 4 TIMES fd(70);rt(90)
```

You end up with the same square as before and save one instruction.

You could of course produce other types of figures for example a triangle.

```
LOOP 3 TIMES fd(70);rt(120)
```

or what about five sides:

```
LOOP 5 TIMES fd(70);rt(72)
```

You could also draw more interesting figures - try :

```
LOOP 20 TIMES fd(70);rt(162)
```

The speed at which the figures are drawn can be increased by removing the turtle.

```
ht
```

(short form for *hide turtle*) you can of course refresh the turtle again by typing :

```
st
```

(short form for *show turtle*).

1.3 Make your own routine.

We have now seen how to produce a number of different figures. We can now produce a small routine (program) that will reproduce the figures as required. (you can make the input window a little larger if you like)

```
0010  // Figures
0020
0030 USE TURTLE                                          (cont.)
```

```
0040
0050 PROC square(s)
0060   LOOP 4 TIMES
0070     fd(s)
0080     rt(90)
0090   ENDLOOP
0100 ENDPROC square
0110
0120 PROC triangle(s)
0130   LOOP 3 TIMES
0140     fd(s)
0150     rt(120)
0160   ENDLOOP
0170 ENDPROC triangle
0180
0190 PROC pentagon(s)
0200   LOOP 5 TIMES
0210     fd(s)
0220     rt(72)
0230   ENDLOOP
0240 ENDPROC pentagon
```

the procedures *square*, *triangle* and *pentagon* have to be made known to the system before we can use them. This is done by typing :

```
SCAN
```

(or by using F6).

Now you can make a pentagon very easily : type :

```
pentagon(60)
```

or a smaller one

```
pentagon(20)
```

1.4 Modules.

The program in section 3: can be saved to disc by typing :

```
LIST "name"
```

where *name* is the name of the disc file, where you want to place your module.

The program will be written to the file in text form and therefore could also be read by a text editor. From AmigaCOMAL you can read the program again as follows:

```
ENTER "name"
```

You can also save your program in code-form by typing:

```
SAVE "name"
```

If you use this form of writing your program to disc, to re-read it you must type:

```
LOAD "name"
```

If you save large programs in code form they are much quicker when you load them again when you want to use them. But the only program that can read this format is AmigaCOMAL.

There is of course a short cut instead of typing LIST, ENTER, SAVE or LOAD you can just type LIST "", ENTER "", SAVE "" or LOAD ""

(you could also use F7, F8, F9 or F10 followed by <RETURN>). Then AmigaCOMAL will produce the necessary file request.

As you see you can save your program and latter read it in again and build extra figures if you like into the program. You can also save the program as a *package*.

You change your program into a package by adding CLOSED to all procedures and adding an export-list to the start of the program.

```
0010  // Figure package
0020
0030 USE TURTLE
0035
0040 EXPORT square, triangle, pentagon
0045
0050 PROC square(s) CLOSED
0060   LOOP 4 TIMES
0070     fd(s)
0080     rt(90)
0090   ENDLOOP
0100 ENDPROC square                                    (cont.)
```

```
0110
0120 PROC triangle(s) CLOSED
0130   LOOP 3 TIMES
0140     fd(s)
0150     rt(120)
0160   ENDLOOP
0170 ENDPROC triangle
0180
0190 PROC pentagon(s) CLOSED
0200   LOOP 5 TIMES
0210     fd(s)
0220     rt(72)
0230   ENDLOOP
0240 ENDPROC pentagon
```

The package is saved as follows:

```
SAVE "Figures.cmp"
```

NOTE the file type must end with "cmp"!

When a program is saved as a package it can be used in the same way as the *Turtle package*. Type NEW to clean the program work area and type the following:

```
USE Turtle
USE Figures
graphicscreen(0)
```

and now f.ex.

```
square(50)
```

You could make it yet easier to use the Figure-package by letting it open the graphic system (*graphicscreen(0)*) and then re-export the central routines from *Turtle*. This could be done as follows: eg.

```
0010  // Figure package
0020
0030 USE TURTLE
0035
0040 EXPORT square, triangle, pentagon
0042 EXPORT fd, rt, cs, ht, st
0045
0047 graphicscreen(0)
0048
```

Now to start you would type:

```
USE Figures
```

Before you use the figure package.

Actually the package *Turtle* is written in AmigaCOMAL. It then calls the assembler coded package *UniGraphics* and re-exports all the routines from there. You can get a lot of good ideas by studying the turtle package and following the same procedures.

1.5 Other packages.

In the catalogue *Packages* all the packages are supplied as standard with AmigaCOMAL. They are all described in the manual you have I hope in front of you. Let's take a look at two of the packages, namely *Iff* and *IntuiSupport*.

The following program reads these packages (and they read other packages so don't worry if there is a lot of disc activity). The program will create a few menus and then go into a loop and wait until you choose one of the menu's.

Choosing one of the first three will cause a picture to be read in and shown in the output window. The last option stops the program (but first after verification).
The program is located in *LstFiles* and can be started by typing ENTER "IntuiSupportDemo".

```
0010  // IntuiSupport and IFF demo
0020
0030 USE IFF
0040 USE INTUISUPPORT
0050
0060  // Create some menus
0070 newmenu("Pictures ",picturenum)
0080 newitem("LOGO       ","1",logonum)
0090 newitem("MandelBrot ","2",mandelnum)
0100 newitem("Polygon    ","3",polygonnum)
0110 newmenu("Stop menu",stopmenunum)
0120 newitem("Stop program ",".",stopnum)
0130
0140 REPEAT
0150   stopprogram:=false
0160   CASE intuiwait OF
0170   WHEN keypressed
0180      // Key pressed - no action in this demo
0190   WHEN mousebottomdown                                    (cont.)
```

```
0200      // Left mouse bottom pressed - no action
0210    WHEN menuselected
0220      CASE menunumber OF
0230      WHEN logonum
0240        CHDIR "IFF_Pictures"
0250        load_window("CbmLogo.iff")
0260        CHDIR "/"
0270        offmenu(logonum)
0280        onmenu(mandelnum)
0290        onmenu(polygonnum)
0300      WHEN mandelnum
0310        CHDIR "IFF_Pictures"
0320        load_window("Mandel256.iff")
0330        CHDIR "/"
0340        offmenu(mandelnum)
0350        onmenu(logonum)
0360        onmenu(polygonnum)
0370      WHEN polygonnum
0380        CHDIR "IFF_Pictures"
0390        load_window("Polygon.iff")
0400        CHDIR "/"
0410        offmenu(polygonnum)
0420        onmenu(mandelnum)
0430        onmenu(logonum)
0440      WHEN stopnum
0450        IF accept("Leave the demo?","Yes","No") THEN
0460          resetmenu
0470          stopprogram:=true
0480        ENDIF
0490      OTHERWISE
0500         // no action
0510      ENDCASE
0520    OTHERWISE
0530       // no action
0540    ENDCASE
0550  UNTIL stopprogram
```

1.6 Editing.

You can add lines to a program simply by typing the line number you wish to add or change. If you are altering larger programs it is very easy to loose track of your position in a program. The best way to make changes is to use *EDIT*.

Let's use this helpful routine to make a few changes and correction to one of the existing programs. First read program *PrintTree* into memory by typing

```
ENTER "PrintTree"
```

(or by using F8). This program shows the structure of a diskette's catalogue. To simplify matters a portion of this routine has been placed in a package named *Directory*. You can execute the routine by typing *RUN*.

As you can see you receive a print out that looks much like the one you would get by typing *dir AmigaCOMAL: all*, only the file names are not printed.

We can now alter the program a little bit so that we can see a graphic representation of the catalogue structure.

Type in *EDIT*. This produce will display the first page of the program and places the cursor in the first line of the program list. You can now use the arrow keys to move the cursor down to line 40 and then press the key combination <shift> + <Alt> + <num 0> (<num 0> is also marked <Ins>). The result should be that line number 0031 is inserted into the correct position in the program.

Type *USE UNIGRAPHICS* and when you have finished move the cursor down one line using the arrow key. The result is that the entire line :

```
0031 USE UNIGRAPHICS
```

is placed into the computer memory.

When you use the command *EDIT* you enter a mode known as (naturally) edit mode. This insures that everything you type on the screen is automatically placed into the program in the computer at the same time, in this way you are always certain that what you see on the screen is identical with the actual program in the computer memory. When you have finished changing a line or typing in a new line you can be certain that this line is also in the program in memory. (After you have moved to another line.)
The instruction *USE UNIGRAPHICS* causes the graphic package *UniGraphics* to be loaded.

Place the cursor on line 0040 again and press <shift> + <Alt> + <num 0> in order to insert a new line (this line becomes 0032). To make the program easier to read we will leave this line blank, so use the arrow key to jump down to line 0040 and press <shift> + <Alt> + <num 0> to insert another new line (this time line 0033) and add *window(0,80,-20,0)*.

Continue to type new lines into the program using the same technique as the first two lines so that the program contains the changes as seen below. Remember that the cursor must be placed on the line where you wish to insert a new line for every new line you insert. (It is not advised to use the mouse in EDIT Mode.)

```
0031 USE UNIGRAPHICS
0032
0033 graphicscreen(0)
0034 window(0,80,-20,0)
0035 clear
0036 moveto(1,-1)
```

All of the lines added to the program are graphical instructions. I will not explain them at the moment as they are all described in the manual in the chapter regarding graphics - UniGraphics.

Using the same method, add the following lines:

```
0131 move(2,-0.5)
0141 dy:=0

0171 draw(0,dy-0.5)
0172 x1:=xcor; y1:=ycor
0173 draw(2,0)

0181 dy:=ycor-y1
0182 moveto(x1,y1)

0201 move(0,dy)
```

When you are finished move the cursor to line 240. Delete that line by typing <shift> + <Alt> + <Del> (The line is removed from the screen and computer memory. Now add a new line instead of the one you have deleted (type <shift> + <Alt> + <num 0>). This line is given number 0231. Type :

```
plottext(xcor+0.5,ycor-0.25,dirname$)
```

and move the cursor to the next line to insure updating in memory.

We have now made the changes we wanted for the time being, so let's see what we have done. First we have to leave the EDIT Mode by typing <right Amiga> + <S> (or just <Esc>). The cursor will now position automatically to an empty line at the bottom of the screen.

Now start the program by typing *RUN* (or use F5).

As you can see the program shows our files with a little more class, but we could make it a little bit more interesting by adding color.

What about a light background, black lines and orange text ?? (blue in Workbench 2.0). Bring AmigaCOMAL into edit-mode by typing *EDIT* and move the cursor down to line 35. Here we add a line using the normal method <shift> + <Alt> + <num 0>.

Notice nothing happened!

The problem is that there are no line numbers available at present. In order to make room for our new lines we will have to re-number the program. Type *Renumber* in the Command menu (The menu on the right - press the right mouse button). The result is seen immediately on the screen.

The cursor has moved down one line so we move it up again (notice the new line number 80). Type in the new line:

```
0071 background(1)
```

(in Workbench 2.0 use 2 not 1).

Now move the cursor down to line 260 and type

```
0251 pencolor(2)
```

(in Workbench use 1 not 2). Now type:

```
0371 pencolor(3)
0372 textstyle(16)
```

OK we are finished ! Leave EDIT-Mode and run the program.

You can naturally look at the contents of other disc's by simply changing the name in line 110 (original line number 50). If there are many catalogues on the disc or sub catalogues it might not be possible to see them all.

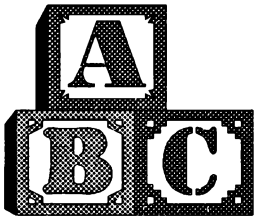
You can make a larger graphic screen by changing lines 0060 and 0070 as follows:

```
0060 graphicscreen(2)
0070 window(0,80,-40,0)
```

and add the lines:

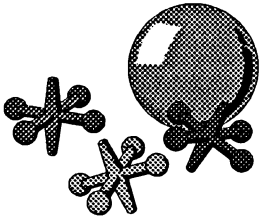
```
0111 WAIT
0112 textscreen
```

The statement *graphicscreen(2)* causes the interlace-graphic-screen to be used. Press any key to return to AmigaCOMALs command window.



TUTORIAL

REFERENCE



REFERENCE

COMPILER

AmigaCOMAL

UniComal

AmigaCOMAL Description

UC 04.90

Copyright Notice

UniComal A/S
H.J. Holst Vej 5 A
DK-2605 Brøndby
DENMARK

Single CPU License

The price paid for one AmigaCOMAL, including a manual and a diskette, licenses you to use the product on one CPU when and only when you have signed and returned the **License Agreement** printed on the last page of the UniComal Reference Manual.

Disclaimer

UniComal A/S has made every effort to supply a dependable product of the highest possible quality. However, UniComal A/S makes no warranties as to the contents of this manual and the system and supplementary diskettes and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. UniComal A/S further reserves the right to make changes to the specifications of the AmigaCOMAL system and the contents of the manuals without obligation to notify any person or organization of such changes. Nevertheless, it is the intention of UniComal A/S to provide all registered users with a periodic newsletter as required providing update information at no charge for a period of one year from the purchase date.

AmigaCOMAL - version 2.10 -
Copyright (C) 1990, 1991
UniComal A/S and UniComal Documentation Center

This document was prepared using WordPerfect ver. 5.1
and a HP LaserJet Series II.

AmigaCOMAL: The Developers System

Produced by

Svend Daugaard Pedersen

AmigaCOMAL: The Developers System Manual

Written by

Svend Daugaard Pedersen

Len Lindsay

Table of Contents

1. Introduction 5

2. Getting Started 7
- 2.1 The Keyboard 7
- 2.2 The Mouse 11
- 2.3 Windows 12
- 2.4 Installing AmigaCOMAL 13

3. - AmigaCOMAL Reference 19

4. - Names in AmigaCOMAL 79
- 4.1 Number Variables 80
- 4.2 String Variables 81
- 4.3 Structured Variables - RECORDs 81
- 4.4 Indexed Variables 84
- 4.5 Pointer Variables 85

5. - Packages 91
- 5.1 Using Packages 91
- 5.2 Programming Packages 92
- 5.3 Routine Libraries 94
- 5.4 Standard Packages 96
- 5.5 System Packages 97
  - 5.5.1 System_code 97
  - 5.5.2 System 99
- 5.6 Interface Routines to the Amiga Libraries 102
  - 5.6.1 Library Packages 102
  - 5.6.2 Support Packages for the Library Routines 103
  - 5.6.3 Messages 103
  - 5.6.4 Devices 105
  - 5.6.5 Screens 107
  - 5.6.6 Windows 108
  - 5.6.7 Gfx 110
  - 5.6.8 Layers 111
  - 5.6.9 Rastport 111
  - 5.6.10 View 112

5.6.11 IntuitionSupport ... 113
5.6.12 Iff ... 114
5.7 Graphics Packages ... 115
5.7.1 Graphics ... 116
5.7.2 Turtle. ... 126
5.8 Exceptions in AmigaCOMAL ... 130

6. - Program development in AmigaCOMAL ... 131
6.1 A Maze ... 131
6.2 A Small Database Program ... 135
6.3 Calling System Routines From AmigaCOMAL ... 147
6.3.1 Sorted Listing of a Directory ... 148
6.3.2 Programming of IO Devices ... 153
6.3.3 A Speech Package ... 157
6.3.4 Some Closing Remarks ... 160

Appendix A. ... 161
The AmigaCOMAL disk. ... 161

Appendix B. ... 163
The file system in AmigaCOMAL. ... 163
B.1 Filenames ... 163
B.2 Program Files ... 164
B.3 Sequential Data Files ... 165
B.4 Random Access Files ... 167

Appendix C. ... 169
Expressions in AmigaCOMAL. ... 169
C.1 Numeric Expressions ... 169
C.2 String Expressions ... 170

Appendix D. ... 173
Screen and screen control codes. ... 173
D.1 The Screen ... 173
D.2 Screen Control Codes / CHR$(x) Results ... 176

Appendix E. ... 179
Error numbers and error texts. ... 179
E.1 Syntax Errors ... 179
E.2 Pre-pass Errors ... 181
E.3 Execution Errors ... 182

E.4 AmigaDOS Errors 185

1. Introduction

AmigaCOMAL is an implementation of the programming language COMAL80 for the Commodore Amiga computer. COMAL80 (COMAL for short) has become an old and well established language. The first versions were made in the early 1970's. At that time programming (in BASIC) was being taught to primary school teachers at the Tonder State Teacher' College, Denmark.

Borge Christensen, at that time teacher at the college, realized that the students wrote poor programs and that the main reason was the language BASIC.

BASIC was indeed easy to use and it offered the user much help during program writing but in BASIC you do not have the possibility to use long and descriptive variable names and you do not have the elegant program structures known from Pascal.

Borge Christensen talked to Benedict Lofstedt, teacher at the University of Aarhus, Denmark about the problems and they agreed that it should be possible to combine the best of BASIC and Pascal. The result was the programming language COMAL (COMmon Algorithmic Language).

Together with some students, Borge made the first implementation of the language in 1974. With the introduction of the micro computers in the late seventies more implementations were made. In 1979 the COMAL standard was defined (COMAL80 Kernal) by a number of computer manufactures and some educational institutions.

AmigaCOMAL contains some minor differences the COMAL80 kernal. The differences are those introduced by UniComal in their implementations of COMAL80 on the IBM PC computer (now included in the Common COMAL standard).

AmigaCOMAL contains some extensions, too. The most important of these extensions are:

* the possibility of dividing programs in modules (packages)
* inclusion of structured variables (RECORDs)
* inclusion of pointer variables

This manual is written for those with some knowledge of COMAL or at least another high level programming language. Most of AmigaCOMAL is explained briefly and only the extensions and Amiga specific details are described more deeply.

If you do not have these prerequisites you are encouraged to read one of the books describing COMAL. For example:

Introduction to COMAL by J William Leary
Beginning COMAL by Borge Christensen
Common COMAL Cross Reference by Len Lindsay

2. Getting Started

The AmigaCOMAL system consists of this manual and one disk. You are encouraged to make a working copy of the disk and use the copy only. Avoid removing the write protect tab from the original disk. In the future, if anything happens to your working copy (so that it is unusable), you can make a new copy from the original. However, please do not give away copies of the disk. It is not public domain or shareware. The disk is not copy protected for your benefit. We trust you to abide by the copyright. You may use this one AmigaCOMAL system on one Amiga computer at a time.

There are two ways to startup AmigaCOMAL after booting your Amiga in your usual manner:

- Put the AmigaCOMAL disk (the working copy) into a disk drive, click twice on the icon of the disk and then click twice on the icon for AmigaCOMAL (the turtle).

- From a CLI or SHELL start AmigaCOMAL with the command:

  <*path*>AmigaCOMAL [*full path:program name*]

To start AmigaCOMAL from the current directory: AmigaCOMAL. To start it from DF1: and have the HANOI program automatically run:
DF1:AmigaCOMAL :programs/hanoi

2.1 The Keyboard

Some of the keys have special functions as described below.

Keys summary:

<Shift> + <Cursor Right>	Moves to end of line.
<Shift> + <Cursor Left>	Moves to start of line.

<Shift> + <Cursor Up> or <PgUp>	Moves to top of window or if already on top and the top line is a program line the previos page is listed.
<Shift> + <Cursor Down> or <PgDn>	Moves to bottom of window or if already on bottom and the bottom line is a programline the next page is listed.
<Alt> + <Cursor Up> or <Home>	List first page of program
<Alt> + <Cursor Down> or <End>	List last page of program
<Ins>	Toggle INSERT mode.
<Alt> + <Del>	Delete line.
<Alt> + <Ins>	Insert line

Note that <Ins>, <PgUp>, <PgDn>, <Home> and <End> are found on the numeric keypad and that the SHIFT-key must be used in addition to the keys shown above.

<Esc> --
In command mode this key works like <Amiga> + <S>. For example, press <Esc> during the execution of a LIST or DIR command to stop the command.

<F?> --
The function keys have numbers from 1 to 10 (i.e, <F1>,<F2>, <F3>...). Each key has a predefined text attached. Press the key and this text will be output on the screen just as if you had typed each character of the text, one by one.

<Del> --
Deletes the character at the cursors position and the rest of the line is moved one character to the left.

<-

The back space key works like the <Del> but instead of deleting the character at the cursor position, the character immediately to the left is deleted. Consequently the key has the same effect as <Left> followed by <Del>.

<Ins> --

Toggles the insert flag. You will move from insert mode to overwrite mode or back again. This key is on the numeric keypad. Use <Shift> + <0> (shifted zero on the keypad).

<Alt> + <Del> --

Deletes the cursor line and scrolls the remaining lines up.

<Alt> + <Ins> --

Inserts an empty line at the cursor position and scrolls the remaining lines down.

<Enter> --

The Enter keys are very important in AmigaCOMAL. Depending upon your computer model, the word ENTER or RETURN or just a broken arrow symbol will be on the keytop. There may be two of the keys on the keyboard, each identical in function. Press one of these keys and AmigaCOMAL reads the contents of the line containing the cursor. In direct mode, if the line contains a command, the command will be executed. If it contains a program line, the line will be checked for correct syntax and then stored. During program execution, pressing this key signifies the end of your INPUT.

<Cursor keys> --

Used to move the cursor. If one of these keys are pressed (unshifted) the cursor moves in the corresponding direction. If the cursor hits the bottom of the window and the bottom line (or previous line) contains a program line, the program will be scrolled down. Likewise moving up past the top.

<Shift> + <Cursor keys> --

Moves cursor to the indicated edge of the window.

<Shift> + <Right>

Goes to the end of the line. If you are already at the top of the screen, <Shift> + <Up> will page the program listing up (if the top line listed is

a program line). Likewise with <Shift> + <Down> at the bottom of the window. <PgUp> and <PgDn> on the numeric keypad (used with <Shift>) work just like <Shift> + <Up> and <Shift> + <Down>.

<Ctrl> + <C> --
Repeats a FIND or CHANGE command. See description of FIND and CHANGE in Chapter 3.

<Amiga> + <S> --
This key combination works much like <Esc>. In command mode the effect is exactly the same: to stop the execution of a command. But in addition, this key combination also stops the execution of a program. <Amiga> + <S> is a short cut for the STOP menu.

<Amiga> + <N> --
A short cut for the NEW menu. The effect is the same as typing the NEW command.

<Amiga> + <R> --
A short cut for the RUN menu. The effect is the same as typing the RUN command.

<Amiga> + <Q> --
A short cut for the QUIT menu. The effect is the same as typing the BYE command.

<Amiga> + <P> --
A short cut for the PAUSE menu. By pressing this key combination (or by selecting the PAUSE menu) a running program will be put into the waiting state until <Amiga> + <C>is pressed (or the Continue menu is selected).

<Amiga> + <C> --
A short cut for the Continue menu. Being in command mode the effect is the same as typing the CON command. If you are in the Pause state the program execution will be resumed.

Immediately after startup the values of the function keys are:

F1	LIST <Enter>
F2	AUTO <Enter>
F3	RENUM <Enter>
F4	Del
F5	RUN <Enter>
F6	SCAN <Enter>
F7	LIST "
F8	ENTER "
F9	SAVE "
F10	LOAD "

2.2 The Mouse

The mouse is an integrated part of the Amiga. By using the mouse, operation is much easier and more logical (at least when you get used to it). We suppose that you are familiar with the basic use of the mouse, such as moving windows, changing size of windows and closing windows. Inside AmigaCOMAL the mouse may be used in this way, too. Also, as in AmigaDOS, the mouse may be used to:

- see the currently invisible part of the windows.
- move the cursor in the command window.

If you have installed the AmigaCOMAL with a large window, say 50 lines with 100 characters each, you cannot see the whole window on the screen. In the lower border (and in the right border) you will see some scroll bars showing the currently visible part of the window. There are three ways to see some other part of the window. You may move the bar using the mouse, you may click the mouse in the empty part of the border or you may click on the arrows at each end of the bar. Try it and note the effect.

In command mode AmigaCOMAL will try to hold the cursor inside the visible part of the window. Thus the scroll bars will move if you try to move the cursor outside the visible part of the window.

To move the cursor with the mouse just point to the place you want the cursor and then press the left bottom.

2.3 Windows

Normally AmigaCOMAL works with two windows: the Command Window and the Execute Window (plus two information windows). While entering commands and program lines the Command Window is active. During the execution of a program (started by the RUN or the CON command) the Execute Window is active. At other times (in direct mode) all output from a command (except RUN or CON) is written in the Command Window even if the command activates a procedure or a function.

The two information windows are the Memory Window and the Error Window. In the Memory Window the number of free bytes is written. The Error Window is used if a syntax error occurs. The Memory window may be closed by the command:

```
MEMWINDOW-
```

It can be reopened by the command:

```
MEMWINDOW+
```

See also the next section.

The windows may be moved around the screen. If the window is closed and reopened the new position will be used.

It is possible to install AmigaCOMAL not to open the Execute Window. In this case all output from a running program will be written in the Command Window.

2.4 Installing AmigaCOMAL

When AmigaCOMAL is started it searches for an installation file named AmigaCOMAL. preferences in the following directories:

current directory
directory containing AmigaCOMAL
DEVS:

If the file is not found, standard installation parameters are used.

You may create an installation file by using the program Install found on the AmigaCOMAL disk. This program will ask some questions which should be answered. For each question the standard value of the parameter is suggested. If you choose that value just hit <Enter>. Press <Help> for a help screen. The meaning of some of these questions (and possible answers) is discussed below:

COMAL DISK NAME
Here you give the name of the disk containing AmigaCOMAL. A colon (:) means the current device.

INSTALLATION FILE NAME
The file :AmigaCOMAL. preferences is suggested. You may choose another path or, if you are making an installation file for a compiled program, even another name (see the Development manual). For Hard Drive users, you may specify the device and subdirectory for its path.

COMMAND WINDOW
(1) Separate screen (2) Workbench screen
Here you may choose the screen where the command window should open: either the workbench screen or a special screen.

You will only be asked for the answer of the following three questions if a special command screen is chosen.

(1) 320 Pixel screen (2) 640 Pixel screen
Choose the width of the screen.

(1) Interlace (2) Non interlace

The height of an interlaced screen is 400 pixels (512 in Europe) and a non-interlaced is 200 (256). Normally an interlaced screen is flickering so the non-interlace is recommended.

Number of bit planes

The number of bit planes determines the number of colors on the screen. Use the following table:

Bit	Planes	Number of colors
1	2	(black and white)
2	4	
3	8	
4	16	
5	32	
6	64	(extra halfbright)

Every bit plane uses a great deal of memory. Further more, your Amiga will work a little bit slower for each bit plane. Some early Amiga computers might not support extra halfbright.

Number of bit planes in window (max. -)

The number cannot be larger than the number given as the answer in the previous question. Normally one bit plane is used in the command window since the scrolling will be faster and you do not need colors in this window.

How many characters per line (mul. of 4)

This number doesn't affect the physical size of the window. If you choose more characters than will fit into one line on the monitor you can use the mouse to see the hidden part of the line (see section 2.2). Maximum is 128.

Number of lines

This number doesn't affect the physical size of the window. If you choose more lines than will fit into one line on the monitor you can use the mouse to see the hidden lines (see section 2.2).

Which font do you want (not checked)

Type in any font you like. The choice is not checked by the install program. This is done by AmigaCOMAL at startup time.

Font height (not checked)

This number is necessary to specify the font completely. If the font selected by the last two questions does not exist, AmigaCOMAL will stop.

Command Window to Front at Program Stop? (y/n)

When a program ends, do you want the Command Window to be immediately pushed to the front? Or do you want the Execute Window to remain in front.

EXECUTE WINDOW

(0) No window (1) Separate screen
(2) Workbench screen (3) Command screen

Choose the screen where you want your Execute Window. It is possible to chose no Execute Window at all (normally this is done only if you are setting up an install file for a compiled program where you supply your own window). Choice number 3 will only be given if you have chosen a separate Command Screen.

The next three questions are only asked if you chose a separate Execute Screen. The meaning is the same as for the Command Screen.

(1) 320 Pixel screen (2) 640 Pixel screen
(1) Interlace (2) Non interlace

Number of bit planes

Number of bit planes in window (max. -)
The number cannot be larger than the number of bit planes in the screen used to hold the window.

How many characters per line (mul. of 4)

This number doesn't affect the physical size of the window. If you choose more characters than will fit into one line on the monitor you can use the mouse to see the hidden part of the line (see section 2.2).

Number of lines

This number doesn't affect the physical size of the window. If you choose more lines than will fit in the window, you can use the mouse to see the hidden lines (see section 2.2).

Which font do you want (not checked)

Type in any font you like. The choice is not checked by the install program. This is done by AmigaCOMAL at startup time.

Font height (not checked)

This number is necessary to specify the font completely. If the font selected by the last two questions does not exist, AmigaCOMAL will stop.

WRITE MODE - Insert mode at startup (y/n)?

Do you want Insert mode ON at the start (or typeover mode)?

Sticky mode (y/n)?

Do you want the mode to stick, or stay in effect until you toggle it (or do you want it to end when you press <Enter>)?

After you answer the last question, a screen with a Command Window, an Execute Window and a Memory Window (it is the size of a title bar) will be opened. You should place the windows as you would like them at startup and then return to the install program by clicking the mouse on the close gadget in the Command Window.

MEMORY

How much memory do you want to alloc. (Kb) ?

This memory is used to hold a comal program, comal packages, variables etc. The standard value is 64Kb. If you are making the install file for a compiled program you may want a smaller area. More than 8Mb (8192Kb) may be allocated (if you have that much memory in your computer).

(1) Window on (2) Window off?

Choose whether you want the Memory Window displayed when COMAL starts. You can use the MEMWINDOW command to change its status.

NUMBER OF OPEN FILES

Maximum number of open files

The AmigaCOMAL can only handle a certain number of open files at the same time.

DIRECTORIES

What is the path for .LST files ?

What is the path for .SAV files ?

What is the path for .PCK files ?

What is the path for .EXT files ?
If AmigaCOMAL cannot find a file in the current directory the search is repeated in the directory given here. The .LST directory is used by the ENTER and MERGE commands, the .SAV directory by the LOAD, RUN and CHAIN commands, the .PCK directory by the USE command and the .EXT directory is used to hold external procedures and functions (EXTERNAL).

What is the name for default tool ?

When a program is saved on disk by the SAVE command an icon for a project is created. The standard value of this "default tool" for this project (the program) is :AmigaCOMAL. If you have a hard disk you may want to change this.

The Install program is a compiled AmigaCOMAL program. The source is included on the disk. It is possible to change this program, perhaps to read in an existing install file and use its values as suggested values instead of the standard values. It is up to you.

3. - AmigaCOMAL Reference

This chapter will provide a quick reference to the built-in AmigaCOMAL keywords. After each keyword (typed in CAPS to allow you to scan them quickly) its syntax is shown, followed by at least one example (in italics). A brief explanation of the keyword is then presented. If a keyword is a function or procedure, AmigaCOMAL does not automatically captialize it for you, as it does the other keywords (ie, AND is capitalized, abs is not).

Notation used for displaying keyword syntax is a modified form of Backus-Naur notation, similar to that used by COMAL Today, COMAL Handbook, and the COMAL Kernal Standard. Generally its rules are:

(a) Normal text not enclosed in < > is typed as shown (you may type in caps or lower case regardless of how the text is shown).

(b) Items enclosed in < > are supplied by the user. The <> are not typed.

(c) Items enclosed in [ ] are optional. If used, do not type the [ ].

(d) Items enclosed in { } are optional and may have several occurances. If used, do not type the { }.

(e) The | means "or". You choose one or the other of the items divided by the |.

(f) All punctuation should be typed as shown, including the ( ).

Also, when presenting examples which mix items that you type and the response from COMAL, things you type will be underlined.
Remember that the examples given are only example lines, not entire programs. They often require other COMAL statements to actually work, such as initialization or procedure and function definitions.

//

```
//
// anything typed here
```

Statement - Anything after // is ignored, allowing comments in programs. In direct mode, // allows you to overtype on a full screen line. After you type the command, just type // and hit return; the rest of the line is ignored, and the command is executed.

ABS

abs(*numeric expression*)

```
PRINT abs(standard'number)
```

Function - Gives the absolute value of the number. Positive numbers and zero are unaffected, while negative numbers become positive.

ACS

acs(*numeric expression*)

```
temp:=acs(num1+num2)
```

Function - Gives the arccosine of the number.

ALLOCATE

allocate(*pointer var*[,*num*])

```
allocate(new@)
```

Procedure - allocates a data area to a pointer variable.

pointer var is a pointer variable declared in a POINTER statement.

num is a number telling AmigaCOMAL (and the Amiga operating system) where in RAM the data area should be placed. The possible values are:

0 the AmigaCOMAL variable stack is used // default
1 public memory
2 chip memory
4 fast memory

If *num* is nonzero the area is taken from the system heap administrated by the Amiga operating system. For further details consult literature of the operating

system. In most cases a value of zero or one is used. The default value of *num* is 0.

The allocated data area may be freed by calling the procedure deallocate.

More about the use of allocate is found in chapter 4.5.

AND

expression AND *expression*

```
IF number>0 AND number<100 THEN
```

Operator - Gives the result of a logical AND of two expressions, as shown by the following table:

AND	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

This is different than most BASICs in which AND is a bitwise operator. For bitwise AND in COMAL see BITAND.

APPEND

OPEN [FILE] *file#,filename*,APPEND

```
OPEN FILE 2,"test",APPEND
```

File Type - Part of the OPEN statement. The sequential file must already exist on disk, and is opened in APPEND mode. New data is written to the file immediately after the existing data.

ARGARRAY$

argarray$(*num*)

```
IF argarray$(-1)="CLI" THEN
```

Array - argarray$ is a constant array containing information about the startup of the program. The array is most often used by compiled AmigaCOMAL programs.

The content of the array (the index starts at -1) is:

argarray$(-1)	The value is either "Workbench" or "CLI" depending on where AmigaCOMAL (or the compiled program) was started.
argarray$(0)	Contains the name of the program (either "AmigaCOMAL" or the name of the compiled program, possibly preceded by a drive or directory specification if one was used when it was started).

If you are working inside the AmigaCOMAL interpreter, -1 and 0 are the only components of the array. If the program is a compiled AmigaCOMAL program there may be further components:

argarray$(1)	first startup parameter
argarray$(2)	second startup parameter (icon tooltypes only) ...
...	...
argarray$(argnum)	last startup parameter (icon tooltypes only)

The parameters are the tooltypes (to be put into an icon via the INFO menu selection of Workbench) if the program was started from Workbench (each tooltype is assigned to an element in the argarray$ array beginning with 1). Or, if the program was started from the CLI, the tail of the command string becomes argarray$(1). argnum is a function that returns the number of parameters (see description of argnum below).

Example: Let us say you have made this little program:

```
0010 FOR x:= -1 TO argnum DO PRINT x;argarray$(x)
0020 WAIT
```

If you run the program from the AmigaCOMAL interpreter, you would get results similar to this:

```
-1 Workbench
 0 AmigaCOMAL
```

Now, if you use the optional compiler, you could compile the program giving it the name: TestPar. Then you could start the program using a parameter from the CLI by the command:

```
TestPar df0:alfa df1:beta
```

You will see the following print out:

```
-1 CLI
 0 TestPar
 1 df0:alfa df1:beta
```

Press any key to end the program. Now, you also could put parameters as tool types in the icon for the program. To get the same results as shown above, you would make one tooltype that contained: df0:alfa df1:beta. However, using icon tooltypes you may separate the two items into two parameters if you wish. First add a tooltype: df0:alfa. Next add another tooltype: df1:beta. Now double click on the icon to start the program. This is the printout:

```
-1 Workbench
 0 TestPar
 1 df0:alfa
 2 df1:beta
```

ARGNUM

```
argnum
IF argnum>0 THEN check'parm
```

Function - returns the number of startup parameters transferred to the program.

The value of argnum is zero if the function is called from the AmigaCOMAL interpreter or any program run under the interpreter.

If argnum is called from a compiled program started from the CLI the value may be zero or one dependent on whether parameters were appended the program name or not.

If argnum is called from a compiled program started from the Workbench by double clicking on the programs icon, the value returned is the number of values put into the tooltypes of the icon of the program (see argarray$ above).

AS

Export *var* [AS *alias* {,*var*[AS -]}

is used to EXPORT a variable, a procedure or a function in a comal package under another name.

Example:

```
EXPORT alfa() AS beta
EXPORT fd(), fd() AS forward
```

For example AS may be used to EXPORT a procedure, a function.

ASN

asn(*numeric expression*)

```
PRINT asn(numb)
```

Function - Gives the arcsine of the number.

AT

PRINT AT *row,col*: [*print list*[*mark*]]
INPUT AT *row,col*[,*len*]:[*prompt*:][*vars*[*mark*]]

```
PRINT AT 1,1: "Section number:"; num;
INPUT AT 10,1,1:"Yes or No? ":reply$
```

Special - Part of INPUT or PRINT statements, specifying a specific location to start at, similar to having a CURSOR statement immediately before a PRINT or INPUT statement. (PRINT AT may also be combined with a USING format.) Remember, the cursor location is specified row then column, similar to finding your seat in a theater. If *row* is 0, it means stay on the current row. If *col* is 0, it means stay in the current column or position in the specified row. Including a comma or semicolon at the end of the statement causes the cursor to remain on the current line, and not go down to the next line. A comma means stay where it is, while a semicolon means space to the next zone, then stay there (default is one space).

ATN

atn(*numeric expression*)

```
PRINT atn(num1+num2)
```

Function - Returns the arctangent in radians of the number.

AUTO

AUTO [*start line*][,*increment*]

```
AUTO 9000
AUTO 100,100
AUTO ,5
AUTO
```

Command - Makes the COMAL system generate line numbers automatically as a program is typed in. Valid line numbers are 1 - 9999. Each line number is always four characters, padded with leading 0's (0030). AUTO begins with the next line number in the sequence if you do not specify a starting line. If you don't specify an increment, 10 is used. If you don't specify a starting line number, 10 is used, unless program lines already exist. Then the line number to start at is the last line plus the increment.

Just hitting *return* after a line number inserts a blank line in the program.

BITAND

argument BITAND *argument*

```
show(bnum BITAND %00001000)
```

Operator - Returns the bitwise AND of the two numbers, similar to AND in BASIC. Binary constants are prefixed by a %. The following binary table shows how BITAND works:

BITAND	00	01	10	11
00	00	00	00	00
01	00	01	00	01
10	00	00	10	10
11	00	01	10	11

BITOR

argument BITOR *argument*

```
PRINT (bnum BITOR flag)
```

Operator - Returns the bitwise OR of the two numbers, similar to OR in BASIC. Binary constants are prefixed by a %. The following binary table shows how BITOR works:

BITAND	00	01	10	11
00	00	01	10	11
01	01	01	11	11
10	10	11	10	11
11	11	11	11	11

BITXOR

argument BITXOR *argument*

```
bnum=(num1+num2) BITXOR %10000000
```

Operator - Returns the bitwise exclusive OR of the two numbers. BITXOR performs the bitwise XOR operation bit by bit on the two numbers. Binary constants are prefixed by a %. The following binary table shows how BITXOR works:

BITXOR	00	01	10	11
00	00	01	10	11
01	01	00	11	10
10	10	11	00	01
11	11	10	01	00

BYE

BYE

```
BYE
```

Command - Exits COMAL and returns to the computer's operating system. You will be asked to confirm the exit from COMAL as a safeguard. You also may click on the close window gadget in the top left corner of the Command Window to exit COMAL. If the program in memory has been modified and not SAVEd, you will be warned (with a system requestor) that all changes will be lost (as with NEW).

CASE

CASE *control expression* [OF]

```
CASE reply$ OF
CASE choice OF
```

Statement - Begins a CASE structure, allowing a multiple choice decision with as many specific WHEN sections as needed. A default OTHERWISE section may be included that is executed if none of the WHEN sections match the condition (which can be either string or numeric). Statement blocks following each WHEN are indented when listed, but the CASE, WHEN and OTHERWISE statements are not. The system will insert the word OF for you if you don't type it. A summary of the CASE decision structure is shown below:

```
CASE selector OF
WHEN choice list
     statement block
     {WHEN choice list
     statement block}
[OTHERWISE
     statement block]
ENDCASE
```

```
CASE eaten OF
WHEN 0
     PRINT "You might starve."
WHEN 1,2
     PRINT "Good meal."
OTHERWISE
     PRINT "I won't pay the bill."
ENDCASE
```

CAT - converted to DIR. See DIR.

CD - a short form of CHDIR. See CHDIR.

CHAIN

CHAIN *filename*

```
CHAIN "menu"
```

Command / Statement - Loads and runs a program from a disk file. The program must have previously been SAVEd to disk.

CHANGE

CHANGE [*line range*] *text1,text2*

```
CHANGE "zz","printout"
```

Command - used to replace a substring (*text1*) in the program with another string (*text2*). If the line number range (*line range*) is specified, the substring is searched for only in that part of the program. Otherwise the whole program is searched. The search is case sensitive.

If the search string is found, the corresponding line is listed with the search string replaced with the new string. Then the system returns to command mode with the cursor placed at the start of the replaced string. You may now make other corrections (or do any thing else) and then press *Enter* to put the changed

line into the program area. If you do not press *Enter* with the cursor on the corrected line, the line will not be changed!

To continue the search press <Ctr>l+<C>. If you press <Ctrl>+<C> without pressing <Enter> first, the line is not changed and the search continues.

CHDIR

CHDIR [*string*]

```
CHDIR "testprog"
CHDIR
```

Command / Statement - short for "CHange DIRectory" and may be further shortened to CD. The string expression, *string*, is the name of a volume or a subdirectory. CHDIR is used to make the directory given by the string expression the current directory. Without *string* the path of the current directory is output (similar to the contents of dir$). The current unit and current directory are the same. So, if as part of a program you type UNIT, it will be converted to CHDIR for you. See CD, DIR$, UNIT and UNIT$

Examples:

`CHDIR "Packages"`	The packages directory will be current directory (this directory must be a subdirectory of the current directory).
`CHDIR ":MyDrawer"`	The directory MyDrawer in the root directory becomes the current directory.
`CD "RAM:"`	Change to RAM: disk
`CD`	Output path of current directory

CHR$

chr$(*numeric expression*)

```
PRINT chr$(num)
```

Function - Returns the character with the specified numeric (ASCII) code. ORD is the complementary function to CHR$.

CLOSE

CLOSE [[FILE] *filenum*]

```
CLOSE FILE 2
CLOSE
```

Command / Statement - Closes the file specified. If no specific file is specified, all files are closed (but does not affect files opened via the SELECT OUTPUT statement). No error occurs if you issue the CLOSE command to close all files if no file is open.

CLOSED

PROC *procname*[(params)] [CLOSED]
FUNC *funcname*[(params)] [CLOSED]

```
PROC newpage(header$) CLOSED
FUNC gcd(n1,n2) CLOSED
```

Procedure / Function Type - Declares that all variables and arrays inside the procedure or function are to be local - hidden from the main program. Likewise, all variables and arrays in the main program are not known inside a CLOSED procedure or function. However, specific variables and arrays may become known inside a CLOSED procedure or function by use of parameters or the IMPORT or GLOBAL statements. Data statements inside a CLOSED procedure or function are considered local. A CLOSED procedure or function may be made EXTERNAL (see EXTERNAL.)

CON

CON

```
CON
```

Command - Restarts a program that was previously stopped by a STOP statement, break key <Amiga> + <S> or the STOP menu.

Due to the internal linking system used by COMAL, if lines are added, deleted, or modified, or if new variables are introduced, the program may not be able to be continued. See also STEP.

COPY

COPY "*source filename*","*destination filename*"

```
COPY "test.sav","final.sav"
```

Command - Copies the specified file. If the file specified is a COMAL program file, its matching icon file is copied also.

COS

cos(*numeric expression*)

```
PRINT cos(number)
```

Function - Returns the cosine of the number in radians.

CREATE

CREATE *filename,# records,record size*

```
CREATE "names",128,200
```

Command / Statement - Creates a random access file of the specified size. Records in the file are numbered beginning with record number 1 (there is no record number 0 in a random access file.)

CURCOL

curcol

```
column:=curcol
```

Function - Returns the current column position of the cursor on the line in the current output window (Command Window in direct mode, Execute Window in a running program). Columns are counted from left to right. The leftmost column is 1.

CURROW

currow

```
row:=currow
```

Function - Returns the current row in the current output window (Command Window in direct mode, Execute Window in a running program). Rows are counted top to bottom. The top row is 1.

CURSOR

CURSOR *line,position*

```
CURSOR 1,1
```

Command / Statement - Positions the cursor to the specified row and column in the current window. Rows are counted from top to bottom; columns from left to right. The top row is line 1. The leftmost column is column 1. Cursor positioning is similar to finding your seat in a theater. First find the row, then the position in that row.

Specifying 0 as a row or column means not to change it, thus CURSOR 0,9 would move to position 9 on the current row.

If used as a direct command, CURSOR correctly positions the cursor, unlike other implementations of COMAL.

DATA

DATA *value*{,*value*}

```
DATA "Sam",134,"Fred",22,"Gloria",46
```

Statement - Declares data constants that may be assigned to variables via a READ statement. Data may be text strings within quotes, or numbers. Multiple items may follow a DATA keyword, separated by commas. When the last DATA item is read, EOD is set to TRUE. Data can be reused following a RESTORE command.

To include a quote mark as part of the string data, use two consecutive quote marks ("abc""defg" is read as abc"defg).

Data inside a CLOSED procedure or function is regarded as local data. Likewise, a READ statement inside a CLOSED procedure or function may only read data inside that procedure or function.

DATE$

date$

```
PRINT "Year:";date$(1:4)
```

Function - returns the date in the format:

```
yyyy-mm-dd          1991-08-27
```

where:

yyyy	the year	1991
mm	the month	08
dd	the date of the month	27

The correct date will be returned only if the Amiga operating system has had the date set correctly via a battery backed up clock, CLI date command, or the date section in Preferences. You cannot set the date with the COMAL function date$.

DEALLOCATE

deallocate(*pointer var*)

```
deallocate(new@)
```

Procedure - deallocates the data area allocated to the pointer variable *pointer var* by allocate.

For further details see chapter 4.5.

DEL

DEL *range*

```
DEL 460
DEL pause
```

Command - Removes (deletes) lines from the program currently in the computer's memory. Lines may be deleted one at a time or in consecutive blocks, all at once.

`DEL 10`	deletes line 10
`DEL` pause	deletes the procedure or function named pause
`DEL 10-30`	deletes all the lines in the range of 10 through 30
`DEL -90`	deletes all program lines up to and including line 90
`DEL 9000-`	deletes all program lines after and including line 9000

DELETE

DELETE *filename*

```
DELETE "test5.sav"
```

Command / Statement - Removes a file from disk. When you delete a COMAL .sav program file, its associated icon file is also deleted.

DIM

DIM *string var* OF *max char*
DIM *str array(index)* OF *max char*
DIM *array name(index)*

```
DIM name$ of 30
DIM players$(1:4) OF 10
DIM scores(min:max)
```

Command / Statement - Allocates (dimensions) space for strings and arrays (AmigaCOMAL extends this to also allow you to DIM numeric and integer variables). Arrays begin with element 1 unless otherwise specified.

Multiple DIMs may be in one statement, separated by commas. Redimensioning is not allowed. However, a DIM may be included in a CLOSED procedure or function. Since the procedure or function is CLOSED, when it finishes executing, all its variables are erased from the system. Thus each time it is called, all DIM statements are treated as being executed for the first time.

Each element of a numeric array is initially set to 0 when dimensioned. Arrays may have multiple dimensions, with whatever top and bottom limits you wish, within memory limitations.

DIR

DIR [*drive/dir*]

```
DIR
```

Command / Statement - Gives a catalog (directory) of the files on a disk. It uses the default directory if none is specified. It may be included in a program.

`DIR "df1:"`	dir of current directory in drive df1:
`DIR "df1:programs"`	dir of directory named programs on df1:
`DIR "programs"`	dir of directory named programs, a subdirectory of the current directory
`DIR ":programs"`	dir of subdirectory named programs, a subdirectory of the root directory
`DIR`	dir of current directory

Both CAT and DIR are included in COMAL, even though they are identical in purpose. This lets you to use the one you are used to. If you type CAT in a program, it is converted to DIR.

DIR$

dir$

```
PRINT dir$
```

Function - returns a string containing the current directory path.

DISCARD

DISCARD

```
DISCARD
```

Command - Discards all previously linked packages and libraries. It is not possible to discard only part of the linked packages.

DISPLAY

DISPLAY [*range*] [TO] [*filename*]

```
DISPLAY "names.lst"
DISPLAY init
```

Command - Lists a program without line numbers. Ranges of lines may be specified, as with LIST. Program lines may be displayed to disk, allowing them to be inserted into word processing documents and such. Program lines DISPLAYed to disk may only be re-entered with the MERGE commands. Press the <space bar> to pause a display. Press it again to continue.

If you cursor up a program listing on the screen, when you hit the top line, COMAL will re-list the previous line, and scroll the listing down. This allows you to backtrack up a listing, and is very handy.

DIV

dividend DIV *divisor*

```
result=guess DIV count
```

Operator - Provides division with an integer answer. It can be used in conjunction with the MOD operator. DIV defines x DIV z as INT(x/z).

DO : see FOR and WHILE

EDIT

Command - This command lists the first page of the program and turns you into edit mode.

Being in edit mode you cannot leave a line that is not a correct AmigaCOMAL line. Each time you try to leave a line by using one of the cursor keys the line is scanned and if correct it is put into the program.

In addition to this:

<Alt> + <Del> deletes the cursor line both in the command window and in the program are.

<Alt> + <Ins> inserts a line with a line number between the surrounding lines and this line is at the same time put into the program area.

As a consequence being in edit mode the lines you see in the command window are the same as those in the program area. The only way to leave a line without putting it into the program is to use the mouse.

You may leave the edit mode by pressing <Esc> or by executing acommand. The RENUM command is an exception from this. This makes it possible to make more rooms between two lines by selecting the Renumber menu. This will renumber the program and the new line numbers will be shown in the window.

ELIF

ELIF *expression* [THEN]

```
ELIF reply$ IN "YyNn" THEN
```

Statement - Allows conditional statement execution. ELIF means "else if" and is part of the IF structure. The statement block following the ELIF is executed only if the condition is TRUE, otherwise it is skipped (the statement block is automatically indented in listings). If you omit the word THEN, the system will insert it for you.

ELSE

ELSE

```
ELSE
```

Statement - Provides alternative statements to execute when all IF and ELIF conditions in the IF structure evaluate to FALSE (the statement block is automatically indented in listings).

END

END [*message*]

```
END  "All Done."
```

Statement - Terminates program execution. END is optional. Without an END statement, a program ends automatically after its last line is executed. There may be more than one END statement in a program. Programs ending at an END statement may not be restarted via CON (use STOP for this capability). If no END statement is used, the program comes to a normal end with no ending message. A message may be included with the END statement. END without a message outputs the word END and the line number. Ending messages appear in the Command Window when the program ends.

ENDCASE

ENDCASE

```
ENDCASE
```

Statement - Marks the end of a CASE structure.

ENDFOR

ENDFOR [*control variable*]

```
ENDFOR sides#
ENDFOR increment
```

Statement - Marks the end of a FOR loop. The system will insert the variable name after ENDFOR for you if you omit it (after a SCAN or RUN). Single line FOR statements do not use ENDFOR.
COMAL will convert NEXT into ENDFOR for you (making the transition from BASIC easier).

The control variable is considered local to the FOR structure. Thus a FOR loop variable will not conflict with a variable of the same name in the main program.

ENDFUNC

ENDFUNC [*function name*]

```
ENDFUNC even
```

Statement - Marks the end of a user defined function. The system will insert the function name after the ENDFUNC for you if you do not type it (after a SCAN or RUN). ENDFUNC is not used with EXTERNAL function header lines (see EXTERNAL).

ENDIF

ENDIF

```
ENDIF
```

Statement - Marks the end of a multi-line IF structure. One line IF statements do not use an ENDIF.

ENDLOOP

ENDLOOP

```
ENDLOOP
```

Statement - Marks the end of a multi-line LOOP structure. One line LOOP statements do not use ENDLOOP.

ENDPROC

ENDPROC [*procedure name*]

```
ENDPROC show'item
```

Statement - Marks the end of a procedure. The system will insert the procedure name after the ENDPROC for you if you do not type it (after a SCAN or RUN). ENDPROC is not used for EXTERNAL procedure header statements (see EXTERNAL).

ENDRECORD

ENDRECORD

```
ENDRECORD
```

Statement - Marks the end of a RECORD structure. See description of RECORD for further details.

ENDTRAP

ENDTRAP

```
ENDTRAP
```

Statement - Marks the end of the error handler TRAP structure.

ENDWHILE

ENDWHILE

```
ENDWHILE
```

Statement - Marks the end of a multi-line WHILE structure. ENDWHILE is not used with single line WHILE statements.

ENTER

ENTER «filename»

```
ENTER "testing.lst"
ENTER "" start file requester
```

Command - Enters program lines from an ASCII format file (such as a file of a program previously LISTed to disk). Any current program is cleared from memory prior to entering the new lines (use MERGE to preserve the current program).

When transferring a COMAL program from one system to another, LIST the program to disk, then ENTER or MERGE it into the other system. This may also be done via modem or networks if the disk format is incompatible.

EOD

eod

```
WHILE NOT eod DO
```

Function - Boolean function that returns TRUE if End Of Data has been reached. If there are no DATA statements in the program, EOD is always TRUE.

EOF

eof(*filenum*)

```
WHILE NOT eof(infile) DO
```

Function - Boolean function that returns TRUE when End Of File has been reached. Since several files may be open at one time, you must specify the file number.

ERR

err

```
CASE err OF
```

Function - Returns the error number when an error occurs within an error handler structure. Error numbers are implementation specific. See Appendix E for a listing of error numbers and their associated messages.

ERRFILE

errfile

```
IF errfile=2 THEN
```

Function - Returns the file number that was in use when the error occurred.

ERRTEXT$

errtext$[(*num*)]

```
PRINT ">>";errtext$;"<<"
```

Function - Used without the parameter this function returns the error text corresponding to the last error if called from inside the HANDLER part of a TRAP .. HANDLER structure. If called from outside of the HANDLER part of a TRAP .. HANDLER structure it will return a null string.

You also have the option to specify which error message you wish to have returned. This is illustrated by the example program.

Example: The following program prints the first fifty error messages on the printer:

```
0010 SELECT OUTPUT "lp:"
0020 FOR x:=1 to 50 DO
0030   PRINT errtext$(x)
0040 ENDFOR x
0050 SELECT OUTPUT "ds:"
```

Note that errtext$ returns exactly the same text as errtext$(err). You may PRINT or assign a substring of errtext$ only after specifying the error number. For example: PRINT errtext$(err)(6:). This will print only the message portion (skips the 4 digit message number). See Appendix E for a listing of error messages.

ESC

esc
TRAP ESC+|-

```
IF esc THEN
TRAP ESC-
```

Function - Returns TRUE if the break key <Amiga> + <S> or STOP menu are detected. This is only useful if break is disabled (via the command: TRAP ESC-). To enable break use the command: TRAP ESC+ . Summary:

`TRAP ESC-`	Disable break
`TRAP ESC+`	Enable break
`PRINT esc`	call to esc function

EXEC

[EXEC] *procname*[(*parameter list*)]

```
show'item(number)
```

Command / Statement - Executes a procedure. May be used from direct mode. The word EXEC is optional and rarely typed. Multiple EXEC statements may be on one line, separated by semicolons. Only the procedure name needs to be typed to execute a procedure. The keyword EXEC is not listed by default. If you want it listed in your program listings, issue the command: SETEXEC+

EXIT

EXIT [WHEN *condition*]

```
EXIT WHEN errors>3
```

Statement - Provides the method for leaving a LOOP structure. It can be conditional with the optional WHEN extension.

EXP

```
exp(numeric expression)
PRINT exp(number)
```

Function - Returns the natural logarithm's base value e raised to the power specified. A good representation of e is 2.718282.

EXPORT

Export *var* [AS *alias* {,*var*[AS -]}

```
EXPORT forward AS fd
EXPORT riskrate
```

Statement - Used in a package to specify all the variables, functions and procedures that are to be exported from the package. The names in the variable list will be known by the program (or package) that uses this package. If AS *alias* is used, then the exported *var* will be known outside of the package as *alias*.

Note that functions and procedures to be exported must be CLOSED. See chapter 5.5.2 for further details and examples.

EXTERNAL

PROC *name*[(*parms*)][EXTERNAL *file*]
FUNC *name*[(*parms*)][EXTERNAL *file*]

```
PROC set'up EXTERNAL "setup.ext"
FUNC rec'size(name$) EXTERNAL "rec.ext"
```

Special - Identifies a procedure or function as an external one. This means that the body of the procedure or function is stored on disk, and is not part of the program itself. Thus ENDFUNC or ENDPROC are not used. An external procedure or function is considered CLOSED. To be used as external, a procedure or function must be CLOSED and previously SAVEd to disk. The filename may be a string variable or string expression.

FALSE

FALSE

```
ok:=FALSE
```

System Constant - Always equals 0. It can be used in comparisons or as a numeric expression. For example, test:=FALSE is the same as test:=0.

FIELD

FIELD *string var* OF *max char*
FIELD *str array(index)* OF *max char*
FIELD *array name(index)*
FIELD *numeric variable name*

```
FIELD name$ of 30
FIELD players$(1:4) OF 10
FIELD scores(min:max)
FIELD filename!(108)
FIELD diskkey#
```

Statement - Used to declare a field within a RECORD data structure. The syntax of FIELD is exactly the same as that of DIM.

See DIM and RECORD for further details.

FILE : see CLOSE, INPUT, OPEN, PRINT, READ, WRITE

FIND

FIND *"text string"*

```
FIND " PROC "
```

Command - Searches the program for specified text. It is case sensitive, so that endif would not match ENDIF. To repeat the FIND command, type <Control> + <C>. If you type <Return> first, the next found line will be underneath the previous one. If you just type <Control> + <C> several times in a row, each found line will overtype the previous one.

FLOAT

float(*numeric expression*)

```
float(5)
```

Function - Returns the number specified as a real number (in its floating point representation).

FOR

FOR *var*:=# TO # [STEP #] DO [*statement*]

```
FOR x:=10 TO 1 STEP -1 DO PRINT x
FOR getin:=1 TO max DO getinput(getin)
FOR num:=1 TO total DO
```

Statement - Marks the start of a FOR structure or one line FOR statement. The variable is initialized to the start value before loop execution begins. A check is made that the variable value does not exceed the end value before executing the loop statements (it is possible for the loop to be skipped entirely if the start value exceeds the end value to begin with). If the step value is negative, the variable is decremented with each loop, rather than incremented. The variable may be an integer variable.

The FOR loop variable is considered LOCAL to the FOR structure. Thus a FOR loop variable will not conflict with a variable of the same name in the main program. A one line FOR statement may be used as a direct command.

The statement block within a multi-line FOR structure is automatically indented when listed. The system will insert the word DO for you if you omit it. A summary of the FOR structure:

FOR *var*:=*start* TO *end* [STEP *amount*] [DO]
  statement block
ENDFOR [*var*]

```
FOR x:=1 TO 12 DO
   PRINT monthname$(x)
ENDFOR x
```

FREE

free

```
PRINT free
```

Function - Returns the amount of free memory available to the COMAL program.

FREEFILE

freefile

```
editfile:=freefile
```

Function - Returns the first free file (stream) number that is available (or 0 if none).

FUNC

FUNC *name*[(*parm*)] [CLOSED]
FUNC *name*[(*parm*)] EXTERNAL *filename*

```
FUNC but'first$(text$) CLOSED
FUNC call'answered EXTERNAL "call"
FUNC occurances#(text$,c$)
```

Statement - Marks the start of a user defined function. Parameter passing is allowed, multiple parameters separated by commas. Parameters used are considered local to the function unless preceded by the REF keyword. If the statement ends with CLOSED, the function is considered a closed function, and all variables and arrays in it are unknown to the main program. Likewise, all variables and arrays in the main program are then unknown to the closed function (except those specified earlier as GLOBAL). Use IMPORT or parameters to bring main program variables or arrays into a closed function.

Every function must include a RETURN statement to return the value of the function. The value may be numeric or string (matching the *function name* type). The block of statements inside the function definition are automatically indented when listed.

Functions may be recursive. You can define a function within another function or procedure (nested). A closed function can be used as an EXTERNAL function by SAVEing it to disk.

Procedures and functions may be the actual parameter of a formal REF parameter. See PROC for an example.

A summary of a function declaration:

FUNC *func name*[({[REF]*parms*})] [CLOSED]
{IMPORT *name*}
 statement block
 RETURN *value* //included in *block*
ENDFUNC [*func name*]

```
FUNC even(num) CLOSED
  IF num MOD 2 THEN
    RETURN FALSE
  ELSE
    RETURN TRUE
  ENDIF
ENDFUNC even
```

GET$

get$(*filenum,# of characters*)

```
text$=get$(2,16)
```

Function - Returns the specified number of characters from the specified file. The file must previously have been opened as a read type file. If the end of file is reached before the specified number of characters are retrieved, only those retrieved prior to EOF will be returned (there is no padding of spaces and no error occurs unless you attempt to read from the file again).

GLOBAL

GLOBAL *var*{,*var*}

```
GLOBAL person@,id#,found,press$(),after(,)
```

Statement - Used to declare certain variables, functions or procedures in the main program as global. A global variable can be seen from all parts of a program including CLOSED functions and procedures.

Sometimes a variable in the main program is IMPORTED into almost all of the closed functions and procedures. In this case the GLOBAL declaration may be used, but the GLOBAL declaration and IMPORT of variables are not always the same.

IMPORT works dynamically. The variable is imported from the environment in which the closed function or procedure is executed (not where it physically is placed in the program). In contradiction to this the GLOBAL declaration is a static property. To see the effect of this let's construct the following package with the name globaltest:

```
0010 EXPORT alfa()
0020
0030 GLOBAL x
0040 x=7
0050
0060 PROC alfa(REF p) CLOSED
0070   PRINT x
0080   p
0090 ENDPROC alfa
```

Store the package to disk with the command:

```
SAVE "globaltest.pck"
```

Now enter the following program:

```
0010 USE GLOBALTEST
0020
0030 GLOBAL x
0040 x=12
0050
0060 alfa(beta)
0070
0080 PROC beta CLOSED
0090   PRINT x
0100 ENDPROC beta
```

The execution of the program will result in the following output:

```
7
12
```

Note that the GLOBAL declaration in the package is unnecessary since all variables in the main part of a package are automatically made global. The declaration is made explicit in the example to be clear.

The use of the static property of a GLOBAL variable is used in the data base example in chapter 6.2.

GOTO

GOTO *label name*

```
GOTO jail
```

Statement - Transfers program execution to the line with the specified label name. Since COMAL has many structures and loop methods, GOTO is not required to be used other than in advanced programs. It is being considered to remove GOTO from the COMAL standard.

HANDLER

HANDLER

```
HANDLER
```

Statement - Marks the beginning of the error handling section of the TRAP .. HANDLER structure. The block of statements in the trapped section and the error handling section are automatically indented when listed.

IF

IF *condition* THEN [*statement*]

```
IF reply$ IN "yYnN" THEN
```

Statement - The start of a multi-line IF structure. May also be a one line IF statement (no ENDIF is used). IF allows conditional statement execution. The block of statements following the IF are only executed if the condition is TRUE. The block of statements are automatically indented when listed. The system will insert the word THEN for you if it is omitted. A one line IF statement may be issued as a direct command.

IF *condition* [THEN]
  statement block
{ELIF *condition* [THEN]
  statement block}
[ELSE
  statement block]
ENDIF

```
IF letter$ IN vowel$ THEN
    PRINT "It is a vowel"
ELIF letter$ IN consonant$ THEN
    PRINT "It is a consonant"
ELSE
    PRINT "It is not a letter"
ENDIF
```

IMPORT

IMPORT *identifier* {,*identifier*}

```
IMPORT running'total
```

Statement - Allows a closed procedure or function to use variables, arrays, procedures and functions from the main program. There may be more than one IMPORT statement in a procedure or function, and all IMPORT statements should come prior to any executable statement in that procedure or function. See also GLOBAL.

IN

string1 IN *string2*

```
IF guess$ IN word$ THEN winner
```

Operator - Returns the position of *string1* within *string2* (or 0 if not found). If *string1* is the null string ("") 0 is returned.

INKEY$

inkey$

```
reply$:=inkey$
```

Function - Returns the next character from the keyboard. It will wait for a key to be pressed (whereas KEY$ does not wait). The key pressed is not printed on the screen (may be useful for entering a password).

INOUT - see SELECT.

INPUT

INPUT FILE *file#[,rec#]: var list*
INPUT [AT *row,col[,len]:][prompt:]vars[mark]*

```
INPUT FILE 2: text$
INPUT AT 5,2,10:"ZIP CODE: ": zip'code,
INPUT "Age? ":age
INPUT reply$
```

Statement - INPUT allows the user to enter data into a running program from the keyboard (the AT section is optional). INPUT FILE gets the data from the file specified, which must have been previously opened for reading. INPUT FILE reads ASCII files, such as those created by PRINT FILE or a Word Processor with ASCII file output (does not read files created by WRITE FILE statements). The prompt is optional and may be a variable.

During the INPUT from keyboard request, the input area is a protected field extending to the end of the line (unless the length part of the AT section is specified). A 0 length means only a carriage return will be accepted. A 0 for the row or column means not to change it (stay in the same row or column). If the *mark* is a comma, the cursor remains where it is after the reply. If it is a semicolon, spaces are printed to the next zone (one space by default if ZONE is not specified), then the cursor remains at that position. See also SELECT INPUT.

INT

int(*numeric expression*)

```
tally:=int(number)
```

Function - Returns the nearest integer less than or equal to the specified number. Both positive and negative numbers are rounded down (-8.3 becomes integer -9).

KEY$

key$

```
WHILE key$<>"c" DO WAIT
```

Function - Returns the first character in the keyboard buffer. If no key has been pressed, the null string ("") is returned. The key accepted by KEY$ is not printed on the screen.

LABEL

[LABEL] *label name*:

```
months:
```

Identifier - Assigns a label name to the line. This label is only referenced by RESTORE or GOTO. It is non-executable and may be placed anywhere within a program as a one line statement. You do not have to type the word label, and if you do, it will not be listed (similar to how EXEC is treated).

LEN

len(*string expression*)

```
length=len(text$)
```

Function - Returns the length of the specified string. All characters, even non-printing characters, are counted. The length of the null string "" is 0.

LET

```
[LET] var:=value          //numeric or string
[LET] var:+value          //numeric or string - incremental
[LET] num var:-value      //numeric only - decremental
count:=5                  // assign a variable a value
name$:="none";sum:=0      // assign several variables separated by ;
reply$:+mark$             // concatenate mark$ onto end of reply$
score:-1                  // decrement score by 1
name$(2):="computer"      // assign an element of array name$
```

Command / Statement - Assigns a value to a variable. The keyword LET is optional and rarely is typed. It is not listed in a program.

LIST

LIST [*range*] [TO] [*filename*]

```
LIST header
LIST "myprog.lst"
LIST pause "pause.lst"
LIST "" starts file requester
```

Command - Lists the specified program lines. If no lines are specified, all lines are listed. If a filename is specified, the lines are listed to that file (in ASCII form), otherwise they are listed to the current output location (screen by default). A procedure or function name can be used to specify a line range.

`LIST 30`	lists only line 30
`LIST -30`	lists all lines up to and including line 30
`LIST 9000-`	lists all lines after and including line 9000
`LIST 100-200`	lists lines 100 through 200 inclusive
`LIST pause`	lists all lines in procedure name pause

Lines LISTed to disk may be merged into the programs with the MERGE command. Statement blocks within structures are automatically indented when listed.

LIST and ENTER commands are useful when transferring programs from one system to another. LIST the program to disk. Then ENTER it into the other system. Modems and networks may also be used to aid the transfer.

Press the <space bar> to pause a display. Press it again to continue. AmigaCOMAL will re-list preceding lines if you cursor up while on the top line of the screen. This is handy to see lines that have just scrolled off the top.

LISTPACK

LISTPACK [*package*]

```
LISTPACK
LISTPACK system
```

Command - Used without the package name *package*, this command outputs a list of all packages linked into the system. When used with the name *package*, it outputs a list of all the procedures in the specified package.

Example: Having USEd the speech package developed in chapter 6.2 the commands have the following effect:

```
LISTPACK
SPEECH (comal)
DEVICES (comal)
MESSAGES (comal)
SYSTEM (comal)
TRANSLATOR_LIBRARY (code)
EXEC_LIBRARY (code)
SYSTEM_CODE (code)

LISTPACK Speech
FUNC translate$(text$) CLOSED
PROC pronounce(text$) CLOSED
PROC say(text$) CLOSED
```

LOAD

LOAD [*filename*]

```
LOAD "menu"
LOAD [""] start file requester
```

Command - Loads a program from disk into the computers memory. Program memory is cleared before loading the program. The program being loaded must have previously been SAVEd to disk. If you do not specify a filename, a requestor will pop up asking for the name. You cannot LOAD a program SAVEd by a different COMAL implementation. To transfer programs between implementations, use LIST to disk, and ENTER to retrieve them.

LOG

log(*numeric expression*)

```
PRINT log(number);
```

Function - Returns the natural logarithm of the number specified. This is log to the base e. A good representation of e is 2.718282.

LOOP

LOOP [*num* TIMES [*simple statement*]]

```
LOOP 10 TIMES
LOOP
```

Statement - The multi-line LOOP structure uses the EXIT statement as the exit method. The statement block within a LOOP structure is automatically indented when listed. Used without the TIMES part (*num* TIMES) the program statements between LOOP and ENDLOOP will be executed again and again without stop unless an EXIT or a GOTO statement directs program execution outside the loop. If the TIMES part is specified, the statements will be executed *num* times (unless interrupted by an EXIT or GOTO statement). The value of the expression *num* must be in the range 1 - 2147483647. A one line LOOP statement may be executed as a direct command. Example:

```
LOOP 20 TIMES fd(70);rt(144)
```

A summary of the LOOP structure:

LOOP [*num* TIMES]
statement block
  [EXIT] // optional
ENDLOOP

```
LOOP
   INPUT"Score(0=done)?":score
   EXIT WHEN score=0
   WRITE FILE 2: score
ENDLOOP
```

MAIN

MAIN

```
MAIN
```

Command - Returns to the main program section. If a program is stopped while an external procedure or function is being executed, COMAL leaves that external routine in memory, available to be LISTed, edited and SAVEd (you may have stopped it to check and fix something). You cannot see the main program in this case! The command: MAIN will return you to the main program, removing the external routine from memory.

MAKEDIR - converted to MKDIR. See MKDIR.

MAXINDEX

maxindex(arrayname({,}))

```
PRINT maxindex(table())
firstone=maxindex(scores(,))
```

Function - Returns the maximum index of the array. If the array is a two dimension array, a comma must be included between the (). A three dimension array needs two commas (,,) and so on.

MEMWINDOW

MEMWINDOW+|-

```
MEMWINDOW+
```

Command - Opens or closes the little Memory Window that continually displays the number of free bytes in the AmigaCOMAL system. These are the two commands:

`MEMWINDOW+` open the free memory window

`MEMWINDOW-` close the free memory window

At the start of AmigaCOMAL the window is closed but this may be changed in the install program. You also may issue the command FREE to see the free memory count (even in a running program) or SIZE for program size information. See FREE and SIZE.

MERGE

MERGE [*line#*[,*increment*]] *filename*

```
MERGE "readrec.lst"
MERGE 580,1 "checkout.lst"
```

Command - Merges program lines from a disk file (ASCII format). The lines are renumbered as they are merged into the current program. If you specify a starting line number, the lines will be renumbered starting at that line as they are merged. If you do not specify a starting line, the merge starts after the last current line. If you do not specify an increment, 10 is used.

MININDEX

minindex(arrayname({,}))

```
PRINT minindex(table())
firstone=minindex(scores(,))
```

Function - Returns the minimum index of the array. If the array is a two dimension array, a comma must be included between the (). A three dimension array needs two commas („) and so on.

MKDIR

MKDIR *string*

```
MKDIR "final"
```

Command / Statement - Creates a new directory (MKDIR is an abbreviation of MaKe DIRectory ... if you type it as MAKEDIR it will be converted into MKDIR). It works like the CLI command MKDIR.

Examples:

`MKDIR ":MyProgs"`	creates a new directory MyProgs in the root of the current directory
`MKDIR "df0:Test"`	creates a new directory Test in the root of drive df0:
`MKDIR "NewDir"`	creates a new directory NewDir in the current directory

A directory which is empty may be removed by using the DELETE command.

MOD

dividend MOD *divisor*

```
rem16=number MOD 16
```

Operator - Returns the modulo of the numbers. It can be used in conjunction with DIV. It defines x MOD z as x-(x DIV z)*z which expands into x-INT(x/z)*z. If z is negative, the result may be irrelevant, but should follow the definition.

NEW

NEW

```
NEW
```

Command - Erases the program currently in memory and clears all variables. Linked packages are also cleared. NEW may be included in a program.

AmigaCOMAL requests confirmation if a NEW command is issued and the program in memory has been modified and not saved.

NEXT : converted to ENDFOR, see ENDFOR

NOT

NOT *condition*

```
IF NOT ok THEN
```

Operator - Returns to reverse of the TRUE / FALSE evaluation:
NOT TRUE = FALSE
NOT FALSE = TRUE

NULL

NULL

```
NULL
```

Statement - This statement does absolutely nothing. It is included in AmigaCOMAL exclusively to be compatible with other COMAL versions. The following statement (or something similar) is sometimes used in COMAL programs for IBM or Commodore 64 to wait for some key (the space key in this example) to be pressed:

```
WHILE key$<>" " DO NULL
```

In a multi-tasking environment like the one you are working in on the Amiga it is bad programming practice to use such a statement to wait for a key to be pressed (it is some times called "busy wait"). AmigaCOMAL will use a lot of CPU time executing this statement again and again. Instead, the following statement should be used:

```
WHILE key$<>" " DO WAIT
```

OF : see CASE, DIM and FIELD

OPEN

OPEN [FILE] *file#,filename,mode*

```
OPEN FILE 2,"scores",READ
```

Command / Statement - Opens a file and assigns it a file number (that is used later with file operation statements). A file may be opened to the screen, printer and serial port as well as disk. The accepted *modes* are:

READ, WRITE, READWRITE, APPEND and RANDOM *length*

OR

condition OR *condition*

```
IF reply$<"a" OR reply$>"z" THEN
```

Operator - Returns the result of the logical OR of the two expressions. This is different than most BASICs in which OR is a bitwise operator. For bitwise OR in COMAL see BITOR.

OR	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

ORD

ord(*string expression*)

```
a=ord("a")
```

Function - Returns an integer representing the ASCII code (ordinal number) of the specified string. If the string is longer than one character, ORD only looks at

the first character. An error results if the null string is used. ORD is system dependent and may vary between systems (especially the Commodore 64 which uses a modified ASCII).

OTHERWISE

OTHERWISE

```
OTHERWISE
```

Statement - Marks the start of the default case in the CASE structure. The block of statements after the OTHERWISE are executed if no WHEN case condition is met. The block of statements are indented automatically when listed. OTHERWISE is an optional part of the CASE structure. If it is omitted and none of the WHEN conditions are met, an error condition will result.

OUTPUT - see SELECT OUTPUT.

PAGE

PAGE

```
PAGE
```

Statement / Command - Clears the screen and puts the cursor at the top left corner (1,1). If output is to another device, a CHR$(12) is sent (form feed).

PASS

PASS *string*

```
PASS "info"
```

Statement / Command - Transfers commands to the CLI.

If the value of the string expression *string* is the empty string a new CLI is opened and you may type CLI commands in this window. To return to AmigaCOMAL in this case, type: endcli

If the value is not empty a window for the output of the command is opened. At the end of the execution of the command you will be asked to press <Enter> to return to AmigaCOMAL.

Examples:

```
PASS ""
```
a new CLI is opened

```
PASS "info"
```
information about the disk system is output

PEEK

peek(*memory address*)

```
device=peek(4839)
```

Function - Returns the decimal value of the contents in the specified memory location. PEEK is very machine dependent. PEEK is a package command in AmigaCOMAL and requires a USE SYSTEM command prior to using it.

PI

pi

```
PRINT "Value of PI is";pi
```

Function - Returns the value of pi. The number of digits varies between systems. AmigaCOMAL sets pi equal to 3.141592654.

POINTER

POINTER *pointer var* [TO *var*]{,*pointer var* [TO *var*]}

```
POINTER len%
POINTER rec'ptr@ TO freezer'article@
```

Statement / Command - Declares a POINTER variable.

Immediately after the declaration, the pointer variable does not point to anything (NIL pointer). To make it point to something an address have to be assigned to the pointer (using the address operator) or a separate data area must be allocated to the pointer (using the procedure allocate).

Example: The following function returns the maximum length of a string variable:

```
FUNC maxlen(REF s$) CLOSED
  POINTER len%
  ^len%:=^s$                    // Make len% point to max. len field of s$
  RETURN len%                   // .. and return content of that field
ENDFUNC maxlen
```

See Chapter 4.5, 6.3.1. for further details and examples.

POKE

poke(*memory address,contents*)

```
poke(4839,13)
```

Function - Places the specified decimal value into the memory indicated memory location. POKEing the wrong value into some memory locations may "lock out" your machine. POKE is very machine dependent.

POKE is a package command, and requires a USE SYSTEM statement prior to using it.

PRINT

PRINT [AT *row,col:*] [USING *form:*] *list*[*mark*]
PRINT [FILE #[*,rec*]:][USING *form:*]*list*[*mark*]

```
PRINT AT 9,1: USING "$###.##": amount
PRINT FILE 2: text$
```

Statement / Command - Prints items as specified. More than one item may be specified in one PRINT statement, separated by a , or a ;. A comma , is a null separator (no spaces between items). A semicolon ; prints spaces to the next zone (one space by default if ZONE has not been specified).

PRINT FILE statements write items in ASCII to the file (it may be preferable to use WRITE FILE for data files). The AT and USING sections are optional parts of a PRINT statement that provide added flexibility. PRINT FILE can write to both sequential and random files. PRINT statements can write to the screen or printer, or even a file if SELECT OUTPUT to a file was previously issued.

PROC

PROC *name*[(*parms*)] [CLOSED]
PROC *name*[(*parms*)] [EXTERNAL *file*]

```
PROC readrec(number)
PROC compare(t1$,t2$) EXTERNAL "comp.ext"
```

Statement - Marks the start of a multi-line procedure definition, including parameter passing (parameters are considered local unless preceded by the REF keyword). A procedure may recursive. The CLOSED keyword is included at the end of the statement to make a procedure closed. A closed procedure does not know about variables or arrays in the main program (unless they are IMPORTed or GLOBAL). Likewise, variables and arrays inside a closed procedure are local, and remain unknown to the main program. A closed procedure can be used as an EXTERNAL procedure by SAVEing it to disk. You can define a procedure within another procedure (nested).

Procedures and functions may be the actual parameter of a formal REF parameter. For example, let the procedures upper() and lower() be two procedures that transforms all letters in a string variable to upper case or lower case letters. These procedures are transferred as parameters in the following little fraction of a program:

```
PRINT "Upper or lower case (U/L)? ",
REPEAT
  answ$:=key$
UNTIL answ$ IN "UuLl"
PRINT
IF answ$ IN "Uu" THEN
  writeline(upper(),line$)
ELSE
  writeline(lower(),line$)
ENDIF
//
PROC writeline(REF convert(), line$)
  convert(line$)
  PRINT line$
ENDPROC writeline
```

Here is a summary of the procedure structure:

PROC *proc name*[({[REF]*parms*})] [CLOSED]
  {IMPORT *name*}
 statement block
ENDPROC *proc name*

```
PROC stars(num) CLOSED
   FOR x:=1 TO num DO
      PRINT "*",
   ENDFOR x
   PRINT // gives cr
ENDPROC stars
```

RANDOM

OPEN FILE *file#,filename*,RANDOM *record length*

```
OPEN FILE 2,"subs",RANDOM 88
```

File Type - Identifies a file as random access, for both reading and writing. Each record in a random access file must be the same length, specified by *record length*.

RANDOMIZE

RANDOMIZE [*seed*]

```
RANDOMIZE
RANDOMIZE 8
```

Statement / Command - Randomizes the random number generator. This generates a series of pseudo random numbers. You only need to use the

RANDOMIZE command once in your program (such as right the the very beginning).

Specify a "seed" number after RANDOMIZE and you cause a specific series of random numbers to be generated. The series of numbers will be the same each time that specific seed is used. This is helpful while testing a program that uses random numbers.

READ

READ [FILE *file#*[,*rec#*]:] *var list*
OPEN [FILE] *filenum*,*filename*,READ

```
READ name$,age
READ FILE 2,record: name$,adr$,city$,st$
OPEN FILE 3,filename$,READ
```

File Type or Statement - In an OPEN statement, specifies a sequential file to be read. READ also can be used as a statement to read data from DATA statements. Finally, READ FILE statements read data from sequential or random files that were created with WRITE FILE statements (these are binary files, not ASCII).

READWRITE

OPEN [FILE] *num*,*filename$*,READWRITE

```
OPEN FILE 3,filename$,READWRITE
```

File Type - In an OPEN statement, specifies a sequential file that can be either read from or written to.

RECORD

RECORD *name@*

```
RECORD persona
```

Structure - The RECORD keyword introduces the RECORD structure used to declare a structured variable. The statements between RECORD and ENDRECORD must be either comment, FIELD, POINTER or a new RECORD structure.

See chapter 4.5, 6.3.1. for further details and examples. The RECORD structure is summarized below:

RECORD *name@*	`RECORD article@`
{FIELD *var info*}	`FIELD name$`
{POINTER *var*}	`FIELD page#`
ENDRECORD *name@*	`ENDRECORD article@`

REF

REF *var*

```
PROC alter(REF text$) CLOSED
FUNC slide(REF text$)
```

Parameter Type - Specifies that the parameter will be an alias for the matching variable or array in the calling statement (passed by reference rather than by value). The value of the calling statement changes as its REF parameter is changed.

RENAME

RENAME *old filename,new filename*

```
RENAME "temp","final"
```

Statement / Command - Renames a disk file. Takes an existing file and gives it a new name. If the file is a .sav COMAL program file, its matching icon file is renamed also.

RENUM

RENUM [*target start*][*,increment*]

```
RENUM 100
RENUM ,5
RENUM 9000,1
```

Command - Renumbers the program in memory. Valid line numbers are 1-9999. By default, it renumbers a program to start at line 10 and increment by 10, unless you specify otherwise.

REPEAT

REPEAT

```
REPEAT
```

Statement - Marks the start of a multi-line REPEAT structure. The block of statements after the REPEAT are automatically indented when listed. They are

continually executed until the condition after the UNTIL evaluates to FALSE. The statements will always be executed at least once. A summary of the REPEAT structure:

REPEAT	`REPEAT`
statement block	`  INPUT "Age: ": age`
UNTIL *condition*	`UNTIL age>0 AND age<110`

REPORT

REPORT [*error code*[,text]]

```
REPORT
REPORT 256
```

Statement - Part of the error handler structure. REPORT causes an error (optionally you can specify what error number to generate). This is useful when using multiple nested handlers. REPORT puts you into the next outer handler. If REPORT is issued while in a trapped section, the error is reported to the handler for that section. If REPORT is issued while not within a TRAP .. HANDLER, the error is reported to the system. REPORT is very system dependent. You also may include a text message along with the error number.

RESTORE

RESTORE [*label*]

```
RESTORE month'names
RESTORE
```

Statement / Command - Allows data in DATA statements to be re-used. The pointer to the next data item is reset back to the first data item, unless a label is specified. Then the next data item pointer points at the first data item following the label (an error occurs if the line immediately after the label is not a DATA line.

RETRY

RETRY

```
RETRY
```

Statement - Used inside the HANDLER part of a TRAP .. HANDLER structure to make AmigaCOMAL re-execute the TRAP part of the structure.

Example: In chapter 6.2. a procedure edittext() in a package EditText will be introduced. The procedure enables you to edit the current content of a string

variable. Using this procedure it is easy to make a procedure enabling you to edit the content of a number variable. Such a procedure may look like:

```
PROC editnum(REF number) CLOSED
  row:=currow; col:=curcol
  TRAP
    t$:=str$(number)
    edittext(t$)
    number:=val(t$)
  HANDLER
    PRINT chr$(7),
    CURSOR row,col
    RETRY
  ENDTRAP
ENDPROC editnum
```

If an invalid character is found in the string edited by the user, the HANDLER part of the TRAP .. HANDLER structure is activated. Here the screen is flashed to signal an error, the cursor is repositioned to the start of the number, and the TRAP part of the structure is re-executed.

The procedure edittext() and a procedure similar to editnum() is in fact used in the install program.

RETURN

RETURN [*value*]

```
RETURN TRUE
RETURN text$
```

Statement - Assigns the value specified after the RETURN to the function and returns control to the calling statement. RETURN may also be used to terminate a procedure early.

RND

rnd [(*start num,end num*)]

```
dice=rnd(1,6)+rnd(1,6)
```

Function - Returns a random number greater than or equal to zero, and less than 1. If start and end limits are specified, RND returns an integer within the specified limits, inclusive.

ROUND

round(*numeric expression*)

```
print round(total)
```

Function - Returns the number rounded to the nearest integer.

RUN

RUN [*filename*]

```
RUN
RUN "menu"
```

Command - Begins execution of the program currently in memory. If a file is specified, the memory is cleared and the file is loaded and run.

RUNWINDOW

RUNWINDOW+|-

```
RUNWINDOW+
```

Command - Turns on or off the Execute Window. Normally all output from a running program is sent to the Execute Window. But by executing the command:

```
RUNWINDOW-
```

the Execute Window is "turned off" and the Command Window will be used as the output window of a running program.

To "turn on" the Execute Window again, execute the command:

```
RUNWINDOW+
```

SAVE

SAVE *filename*[,i|n]

```
SAVE "zombies"
SAVE "" starts file requester
```

It is possible to create an icon for a SAVE file.

The command

```
SAVE "name",i
```

will always create an icon and the command

```
SAVE "name",n
```

will not create an icon.

In both commands it is independent on how AmigaCOMAL has been installed.

The command

```
SAVE "name"
```

will create or not create an icon dependent of how AmigaCOMAL has been installed.

Command - Stores the program in memory to the specified file in compressed form. Comments are not removed. Later the program can be retrieved with the LOAD, RUN, or CHAIN command. If a filename is not specified, a requestor will pop up. Procedures or functions stored with the SAVE command can be used as EXTERNAL procedures and functions.

A SAVEd program may not be transferred to another COMAL system. To transfer a program it must be in ASCII form as with the LIST to disk command. Use the ENTER command to retrieve it to the other system. Modems or networks may also be used to transfer the file.

AmigaCOMAL will rename a file found with the same name (adding the .BACKUP extension) and then save the program (along with an ICON image for it).

SCAN

SCAN

```
SCAN
```

Command - Scans the program in memory for structure errors. Once a program has been SCANned or RUN procedures and functions may be called from direct mode.

SELECT

SELECT *direction type*

```
SELECT OUTPUT loc$
SELECT INPUT infile$
SELECT INOUT "sp:"
```

Command / Statement - Allows you to select the IO (Input / Output) units of your program. Normally the output from PRINT statements in a program is sent to the screen ("ds:") and the INPUT statement gets its input from the keyboard ("kb:").

To select another output unit, use the command:

SELECT OUTPUT *device name*

To select another input device, use the command:

SELECT INPUT *device name*

To select another device for both input and output (for instance "sp:"), use the command:

SELECT INOUT *device name*

The *device name* may be either a file name or the name of a standard device. Examples:

```
SELECT OUTPUT "ds:"          Data Screen
SELECT OUTPUT "lp:"          Line Printer
SELECT INOUT "sp:"           Serial Port
SELECT OUTPUT "filename"
```

The SELECT command will only affect a running program (even if it is executed as a command) unless you have turned off the Execute Window with the RUNWINDOW command.

See also Appendix B.

SGN

sgn(*numeric expression*)

```
flag=sgn(number)
```

Function - Returns -1 if the number is negative. Returns 1 if the number is positive. Returns 0 if the number is 0.

SIN

sin(*numeric expression*)

```
plot(sin(num),y)
```

Function - Returns the sine of the number in radians.

SIZE

SIZE

```
SIZE
```

Command - Displays the amount of available free memory. Some COMALs display more information, such as program size and data area size. This is an informational display only. See FREE for a function that may be used in a program.

SPC$

spc$(*number of spaces*)

```
PRINT spc$(39)
```

Function - Returns the number of spaces specified.

SQR

sqr(*numeric expression*)

```
root=sqr(number)
```

Function - Returns the square root of the number.

STEP

STEP *numeric expression*

```
FOR x=1 TO max STEP 2 DO
```

Part of FOR statement - Sets the amount the FOR variable is incremented after each loop. If it is negative, the loop variable is decremented rather than incremented, and terminates when the variable value is less than the end amount. The step amount can be an integer or real numeric expression.

Note: a non-integer step size can lead to some "round off" problems due to the way addition is performed on the numbers.

STEP

STEP

```
STEP
```

Command - Used in connection with the TRACE command. Having turned on the trace mode using the TRACE+ command, the STEP command will execute the next line in the program, output the values of all calculated expressions, list the next line to be executed and then return to execute mode.

The key combination <Amiga> + <Y> may be used as short cut for STEP.

See TRACE for a further description.

STOP

STOP [*message*]

```
STOP "now inside PROC remove'blank"
```

Statement - terminates program execution. Execution may be continued with the CON command. Variables may be displayed or changed before continuing. Lines may also be listed. However, if any lines are added, deleted, or modified the program may not be able to be restarted (due to internal tables).

STR$

str$(*number*)

```
zip$=str$(number)
```

Function - Returns a string that is the equivalent of the number. The number 567 becomes "567". The VAL function does the reverse, converting a string into a number.

TAB

TAB(*column number*)

```
PRINT TAB(col), name$
```

Function - Prints spaces up to the column specified. If that position is already exceeded, it goes to the specified position on the next line. TAB is always part of a PRINT statement.

TAN

tan(*numeric expression*)

```
PRINT tan(number)
```

Function - Returns the tangent of the number in radians.

THEN

THEN

```
IF NOT ok THEN RETURN FALSE
ELIF errors>3 THEN
```

Part of IF - Part of the IF and ELIF statements. The system will insert the word THEN for you if you omit it.

TIME$

time$

```
PRINT time$
```

Function - Returns the time of the day in the format:

hh:mm:ss	11:48:09

Where:

```
hh      is the hour     (11)
mm      is the minutes  (48)
ss      is the seconds  (09)
```

This function only returns the correct time if the Amiga internal clock is correctly set (either from a battery backed up clock, with the CLI command date, or via the time setting in Preferences).

TIMER

timer

```
starting=timer
```

Function - Returns the content of a constantly running timer in the AmigaCOMAL system. The time is measured in seconds with a resolution of approximately 0.02 seconds (1/60 seconds in USA and 1/50 seconds in Europe). The timer is reset to zero at the startup of AmigaCOMAL.

TIMER

TIMER *num*

```
TIMER 0
```

Statement / Command - Sets the timer in the AmigaCOMAL system. After the execution of the statement the timer is set to the value of the numerical expression *num*.

TO

start num TO *end num*

```
FOR x:=1 TO 4 DO
```

Part of FOR - Part of the FOR statement, separating the start and end numbers.

TRACE

TRACE[+|-]

```
TRACE-
TRACE+
TRACE
```

Command - TRACE is a very useful command helping you to debug your program. There are several versions of the command. The command: TRACE (without using a + or -) has the same effect as RUN but during the execution of the program each executed line and the value of calculated expressions in the line is listed in the Command Window.

Example: The program calculates the least common divisor of 25 and 15 using Eulers algorithm:

```
0010 FUNC euler(m,n) CLOSED
0020   IF m MOD n=0 THEN
0030     RETURN n
0040   ELSE
0050     RETURN euler(n,m MOD n)
0060   ENDIF
0070 ENDFUNC euler
0080
0090 PRINT euler(25,15)
```

If it is started by the TRACE command it will produce the following output in the Command Window:

```
0010 FUNC euler(m,n) CLOSED
0080
0090 PRINT euler(25,15)
<25><15>
0020   IF m MOD n=0 THEN
<0>
0050     RETURN euler(n,m MOD n)
<15><10>
0020   IF m MOD n=0 THEN
<0>
0050     RETURN euler(n,m MOD n)
<15><10>
0020   IF m MOD n=0 THEN
<1>
0030     RETURN n
<5><5><5><5>
```

The final value (5) is printed in the Execute Window, as it normally would if the program was RUN.

Another form of the TRACE command is:

```
TRACE+
```

Having executed this command the system will be set in trace mode which allow you to single step through the program.

While in trace mode the RUN command will do all the usual initialization (clear variables, scan the program, etc.). Then the first line of the program will be listed in the Command Window and the system will return into command mode.

To actually execute the line you have to use the STEP command (or the short cut <Amiga>+<Y>. When a STEP command is issued, the values of all expressions calculated during the execution of the line will be displayed and the next line listed before the system returns to command mode again. By pressing <Amiga>+<Y> several times you will gradually produce the same output as the TRACE command, but at your own rate.

The trace mode may be turned off by executing the CON command, which will continue the program execution in normal mode, or by executing the command:

```
TRACE-
```

TRACE commands execute an implicit RUNWINDOW+.

TRAP

TRAP

```
TRAP
```

Statement - Marks the start of the error trap structure. Inside the HNADLER section, several built in functions can assist you: err, errfile and errtext$. You also may use REPORT or RETRY statements inside the HANDLER section. Error handlers may be nested. The system is like a perculator. An error trapped can be reported to an outer handler (via REPORT). That handler can either deal with it or REPORT it to its outer handler. And so on until the error hits the outside level, where the program would stop with an error message. The *trapped statment block* and *handler statement block* are indented when listed. A summary of the TRAP structure:

TRAP
trapped statement block
HANDLER
handler statement block
ENDTRAP

```
TRAP
   INPUT "Enter age: ":age
   IF age<0 or age>110 THEN REPORT 999
HANDLER
   IF err=999 THEN
      PRINT "Be reasonable."
   ELSE
      PRINT "-";errtext$(err)(6:)
      PRINT "Enter number of years."
   ENDIF
RETRY
ENDTRAP
```

TRAP ESC

TRAP ESC+|-

```
TRAP ESC-       // disable stop/break key
TRAP ESC+       // enable stop/break key
```

Statement - Used to disable / enable the break key or STOP menu. Be careful when disabling a break. If your program gets into a loop, you will not be able to stop it.

TRUE

TRUE

```
RETURN TRUE
```

System constant - Always equal to 1 when used as assignment. Other times it means not FALSE (a value that is not equal to 0).

UNIT

UNIT [*string expression*]

```
UNIT data'drive$
```

Command / Statement - Sets the default unit. Remember that the Amiga refers to disk drives by the name of the disk in them. Thus if you put a disk name AmigaCOMAL into drive df1: this would happen:

```
UNIT "ram:"
PRINT unit$
RAM:
UNIT "df1:"
PRINT unit$
AmigaCOMAL:
```

In a program, UNIT *name* is converted to CHDIR *name* and UNIT is converted to CHDIR. See also UNIT$ and CHDIR.

UNIT$

unit$

```
current$:=unit$
```

Function - Returns the current unit name. The unit can be set with the UNIT command. For an example see UNIT.

UNTIL

UNTIL *condition*

```
UNTIL reply$="q"
```

Statement - Marks the end of the REPEAT structure. Statements inside the structure are executed until the condition is TRUE.

USE

USE *package* [FROM *directory$*]

```
USE system
USE system FROM "AmigaCOMAL:"
```

Statement / Command - Links a package into the AmigaCOMAL system (if the package is not already linked) and makes the EXPORTed names in the package known.

If the package with the name *name* is not already linked, AmigaCOMAL searches for the following files:

```
name.cmp
Packages/name.cmp
name.pck
packages/name.pck
```

or - If the FROM part is present, the following files:

```
directory/name.cmp
directory/name.pck
```

The directory "Packages" is the directory set in the install program (standard setting : ":Packages"):

See chapter 5. for further details.

USING

PRINT [AT *row,col*:]USING *format$*: *var list*

```
PRINT USING "##> $###.##": x, cash(x)
PRINT AT 8,5: USING "##": item
```

Special - Part of a PRINT statement allowing formatted output. Within the format string a # reserves a position for each possible digit of the number; a period "." marks the decimal point location; a minus sign "-" is optional reserving a position for the negative sign. On the right of the decimal point, zeroes are padded where necessary. On the left of the decimal point, spaces are padded. All other characters (other than # . -) are printed as supplied. If the number has more digits than reserved, a * is printed in each reserved position.

VAL

val(*numeric string*)

```
age=val(reply$)
```

Function - Returns the numeric value of a numeric string. This allows you to input data as strings, check them for errors, then convert them into numbers. VAL accepts the digits, + and - signs, decimal point, and exponential notation. Leading spaces are ignored by VAL. Requesting a VAL of a non-numeric string results in an error.

VAL only checks the first part of a string. If it finds a number, it ignores the remainder of the string. Thus val("33 tires") would give a result of 33. However, val("tires 33") would result in an error.

VARSIZE

varsize(*var*)

```
temp:=varsize(name$)
```

Function - Returns the number of bytes occupied by the data field of the variable *var*. The variable may be a simple variable (number or string), an array, a record or even a pointer.

Example: This is a small program:

```
x:=0
s$="anything"
DIM array(10,10)
PRINT varsize(x)
PRINT varsize(s$)
PRINT varsize(array(,))
```

When RUN it will produce the output:

```
8       <--   a real number uses 8 bytes
86      <--   a string uses 5 bytes plus its DIM size rounded up to the
              nearest even number.
800     <--   a 10 by 10 array has 100 elements of real numbers
```

WAIT

WAIT [*num*]

```
WAIT 2
WAIT
```

Statement / Command - Without the number expression *num* the WAIT statement will put AmigaCOMAL into the waiting state until an event happens. This event may be a key pressed, the selection of a menu or pressing the mouse botton.

With the number expression *num* present the execution of the WAIT statement will put the AmigaCOMAL into the waiting state for a number of seconds given by the value of the number expression *num*.

Being in the waiting state, AmigaCOMAL will not use CPU time and thus it will not slow down other processes running in the Amiga.

Example: This program outputs the time used by the LOOP (approximately 4 seconds):

```
0010 WAIT 1
0020
0030 starttime:=timer
0040 LOOP 10000 TIMES x:=10/3
0050 PRINT USING "Time used: ##.##": timer-starttime
```

The WAIT statement in such a program is necessary. At the start of the program the Execute Window is brought to front, but since the Amiga is a multi-tasking machine this is done in parallel with the execution of the AmigaCOMAL program. The WAIT statement ensures that this rearrangement of the windows has been completed before the starting time is read.

In some cases a compiled AmigaCOMAL program will return to CLI or Workbench before you have had time to read the output. In such a case you should terminate the program with a WAIT statement.

WHEN

WHEN *list of values*

```
WHEN "Jan","jan"
WHEN 1,2
```

Statement - Provides a specific case within a CASE structure. One or more values are listed after the WHEN. If any match the current case value, its following statements are executed. The values specified after a WHEN must be of the same type as the one in the CASE statement. Statement blocks are indented automatically when listed.

AmigaCOMAL also allows WHEN to be used as a conditional EXIT from the LOOP structure:

EXIT WHEN *condition*

WHILE

WHILE *expression* [DO] [*statement*]

```
WHILE NOT EOF(infile) DO process
WHILE errors<3 DO
```

Statement - Marks the start of a multi-line WHILE structure. As long as its condition is true the statements are executed. If the condition is FALSE right at the start, the statements are skipped over. The system will insert the word DO for you if you do not type it. The statements inside the WHILE structure are indented automaticall when listed.

A one line WHILE statement is allowed (and may be issued from direct mode). It does not use an ENDWHILE. For example (after opening a text file in direct mode as file number 2):

```
WHILE NOT EOF(2) PRINT get$(2,1),
```

A summary of the WHILE structure:

WHILE *condition* [DO]
statement block
ENDWHILE

```
WHILE NOT EOD DO
  READ num
  sum:+num
ENDWHILE
```

WRITE

WRITE FILE *file#*[,*rec#*]:*var*
OPEN [FILE] *filenum*,*filename*,WRITE

```
WRITE FILE 2: name$
OPEN FILE 3,"scores",WRITE
```

Statement / Command / File Type - Writes data to a file in binary form (not ASCII). As a file type, it specifies that the file is to be written to. Data written by a WRITE FILE statement may be read with a READ FILE statement.

Data files created by WRITE statements are not compatible with other systems. If you wish to transfer data to another system (such as IBM) you must use PRINT FILE statements (which are retrieved with INPUT FILE).

ZONE

ZONE [*tab interval*]

```
ZONE 5
z:=ZONE
```

Command / Statement / Function - Returns the current zone setting as a function. As a statement it sets the zone to the number specified. The zone determines the interval in tab positions on an output line. The default zone is 1 (a zone at each column). The semicolon is the zone separator (for PRINT statements and at the end of INPUT statements). By default, a semicolon outputs one space (if a ZONE statement is previously used, it outputs spaces to the next zone). When a comma is used as a separator in a PRINT statement (or at the end of an INPUT statement) no spaces are printed, and the cursor remains where it is (null separator).

4. - Names in AmigaCOMAL

In AmigaCOMAL, variables, procedures and functions may be named. Names are built up by one or more of the following characters:

letters	abc...xyz
digits	012...789
apostrophe	'
underscore	_

The first character must be a letter and the maximum length of a name is 127 characters. Letters may be international as well as national letters (for instance the Danish letters Æ and æ).

In certain names the last character is a special character used to indicate the type of the named quantity. These characters are:

$	string
#	long integer (32 bits)
%	short integer (16 bits)
!	byte integer (8 bits)
@	record

If those characters are used they will be part of the name.

Special Note: two names such as a# and a$ are treated as different. This is an improvement over IBM and Commodore 64 COMALs.

If a name is found during the execution of a line, AmigaCOMAL searches for information about that name in the environment of the line being executed (for instance a closed procedure). If the information is not found, the search is continued among the global names (a name in the main program may be declared global with the keyword GLOBAL).

4.1 Number Variables

In AmigaCOMAL there are four different types of numbers: real numbers (sometimes called floating point numbers) and three types of integers. The type of a number variable is specified in the name as shown in the previous section.

The value of a real number is either zero or a floating point number with 9-10 digits accuracy and a tens exponent between -9864 and 9864.

Number variables are assigned values in an assignment statement, INPUT statement or READ statement. If the variable does not exist (has not been used before) it will automatically be created. It is possible to declare variables explicitly in a DIM statement, too. Examples of such declarations are:

```
DIM x,y,z          // Declare as REAL number variables
DIM a#,b!          // Declare as INTEGER variables
```

Such explicit declarations may be used inside a closed procedure to cover global variables in the main program.

Although there are four different types of variables, they may be used freely in expressions. Thus the following statement is valid:

```
a%:=2*x-n#/b!
```

AmigaCOMAL keeps track of the types and makes the necessary conversions between the types. In the example above, AmigaCOMAL will calculate the expression. The value is a real number since there is a division in the expression. Before the result is stored in the variable named a% (a short integer variable), the number calculated is rounded to the nearest integer.

Calculations are not executed faster using integer variables in place of real variables. The only benefit in using integer variables is that they occupy less space in RAM and that they make it easier to call routines in the Amiga operating system.

Beginners are encouraged to use real variables only.

4.2 String Variables

In AmigaCOMAL you may work with string variables and string constants. As mentioned earlier, the name of a string variable must end with a dollar sign ($).

Like number variables, string variables are assigned values in an assignment statement, INPUT statement or READ statement. If the variable does not yet exist, it will automatically be created. Such an automatically created string variable is able to contain text strings of up to 80 characters (an automatic DIM of length 80).

If such a size does not fit your needs you should first declare it in a DIM statement where you specify the maximum number of characters. Examples of DIM statements creating string variables are:

```
DIM name$ OF 30, address$ OF 25
DIM text$ of 100
DIM answer$ OF 1
```

The maximum size that may be specified in a DIM statement is 31767 characters.

4.3 Structured Variables - RECORDs

In several connections it would be natural to group together different variables. For instance, making a data base containing information about the contents of your home freezer. For each article in the freezer you might store information about the sort of the article, the quantity, the date of freezing and the last date of use. Thus you need four variables:

```
content$
quantity
date_of_freezing$
date_of_use$
```

Quantity is chosen as a number variable and the others as string variables.

In AmigaCOMAL it is possible to group together such variables in a special data structure called a RECORD.

A RECORD is created by using a special RECORD structure. For instance, we might have created a RECORD for the contents of an article in the freezer in the following way:

```
RECORD freezer'article@
  FIELD content$ OF 20
  FIELD quantity
  FIELD date_of_freezing$ OF 9
  FIELD date_of_use$ OF 9
ENDRECORD freezer'article@
```

As shown in the example, the name of a RECORD must end with a @. The variables in the data structure are created by a FIELD statement with a syntax exactly like the DIM statement.

After the execution of a block of statements like the one shown, the data structure is created and initialized, i.e. numbers are assigned the value zero and strings are assigned the empty string.

You may refer to each field in the RECORD by writing the name of the RECORD separated by a period, for instance:

```
freezer'article@.content$:="Steak"
INPUT "Type in the weight: ": freezer'article@.quantity
PRINT "Last date of use: ",freezer'article@.date_of_use$
```

A field description like freezer'article@.quantity is treated as a normal variable (in this case a number variable) and may be used just like other variables of that type.

The idea of creating a RECORD instead of four single variables is that you may refer to all four by using only a single name.

Let us say you have made a procedure to print out all the fields:

```
PROC output(article@)
```

This procedure may be called with a single parameter:

```
output(freezer'article@)
```

You may likewise write the whole content to a file by using only one variable:

```
WRITE FILE 3: freezer'article@
```

Likewise, you may read it back with only one READ statement:

```
READ FILE 3: freezer'article@
```

A structured variable like freezer'article@ is not only a variable but may be used as a model for creating other structured variables with the same internal structure. This is done in a DIM statement:

```
DIM new'article@ OF freezer'article@, article2@ OF freezer'article@
```

If two structured variables use the same model, you may assign one of them to the other in a statement like:

```
new'article@:=freezer'article@; article2@:=new'article@
```

The fields in a record may be records themselves. These inner records may be declared explicitly by using a RECORD statement inside the other RECORD statement or by using a FIELD statement.

The date in our freezer example could be a record as:

```
RECORD date@
  FIELD year
  FIELD month
  FIELD day
ENDRECORD date@
```

and our freezer'article@ may now be defined as:

```
RECORD freezer'article@
  FIELD content$ OF 20
  FIELD quantity
  FIELD date_of_freezing@ OF date@
  FIELD date_of_use@ OF date@
ENDRECORD freezer'article@
```

The year of freezing may be accessed in the following way:

```
PRINT freezer'article@.date_of_freezing@.year
```

In Chapter 6. you will find several examples of the use of records.

4.4 Indexed Variables

An indexed variable (sometimes called an array) consists of a number of elements organized in a table with one or more dimensions. The elements in the table must be of the same type, i.e. either strings, numbers or records.

An indexed variable is declared in a DIM statement where the type and the number of dimensions are specified. Examples of such declarations are:

```
DIM a(50)              // A 1-dimensional number array with 50 elements
DIM b!(4,3)            // A 2-dimensional byte array with 12 elements
DIM table$(100) OF 25  // An array of strings
DIM freezer@(50) OF freezer'article@   // An array of RECORD's
```

In all the examples the index is numbered from one up to the number stated in the dimension. The elements in the b! array are:

```
b!(1,1) b!(1,2) b!(1,3)
b!(2,1) b!(2,2) b!(2,3)
b!(3,1) b!(3,2) b!(3,3)
b!(4,1) b!(4,2) b!(4,3)
```

You may declare indexed variables having indices starting with a value other than one. For instance:

```
DIM n#(-3:10)
```

It's elements are;

```
n#(-3), n#(-2), n#(-1), ........ , n#(10)
```

Each element in an array is treated as a normal variable (of that type) and may be used like such variables:

```
table$(23):="this is a test"
FOR i:=1 TO 4 DO
   FOR j:=1 TO 3 DO
      READ b!(i,j)
   ENDFOR j
ENDFOR i
PRINT freezer@(12).content$
```

It is possible to fill a whole array with the same value in one assignment statement:

```
table$():="nothing here"
freezer@():=article2@
n#():=0
```

Likewise you may write a complete array to a file in one WRITE statement and read it in one READ statement. The following two program segments do the same thing:

```
FOR i:=1 TO 4 DO
   FOR j:=1 TO 3 DO
      READ b!(i,j)
   ENDFOR j
ENDFOR i
DATA 1,2,3,4,5,6,7,8,9,10,11,12
```

and

```
READ b!(,)
DATA 1,2,3,4,5,6,7,8,9,10,11,12
```

4.5 Pointer Variables

You may create a variable implicitly using an assignment statement like:

```
x:=27
```

AmigaCOMAL will allocate place for the content (the data field) of the variable somewhere in the storage of the Amiga. In the example, the value 27 is then placed in the data field.

With the help of the address operator (^) it is possible to get information of the address of the data field. It is done in this way:

```
PRINT ^x
```

where the printout could be:

```
12982432
```

If you create another variable y, the data field of this variable will be put in another place and the printout from:

```
PRINT ^y
```

would be another number.

Once AmigaCOMAL has placed the data fields of the variables, the addresses will be the same "forever" (until they are cleared and created once more in a new run or in a new call to a closed procedure). An assignment statement like:

```
y:=x
```

copies the content of x to that of y and now there are two copies of this number in the Amiga. The addresses of x and y remain unchanged.

AmigaCOMAL supports another type of variable called a pointer variable where you can change the address of the variables data field. A pointer variable is declared in a special POINTER statement like:

```
POINTER ptr              // A real pointer variable
POINTER p$               // A pointer to a string
```

After the execution of a POINTER statement, the address of the pointer variable is zero, which means that there is no data field connected to the variable (the variable does not point to anything).

You may make the pointer variable point to something by executing a statement like:

```
^ptr:=^x
```

After the execution of this statement the pointer variable ptr points to the content of the variable x. The two PRINT statements:

```
PRINT x
PRINT ptr
```

will result in exactly the same printout. However, the content of the variable x is only found once in the Amiga and if you assign the variable ptr to another value:

```
ptr:=78
```

the value of the variable x will be changed at the same time!

You should note the difference between the statements:

```
ptr:=1256+38
```

and

```
^ptr:=1256+38
```

In the first one, the variable ptr is assigned the value 1294. The second statement makes the pointer point to the content of the address 1294 (and this content is probably unknown).

It is hard to see the meaning of all this. So let us have a look of the following little function:

```
FUNC peek(address#) CLOSED
   POINTER b!
      ^b!:=address#
   RETURN b!
ENDFUNC peek
```

In the first statement of the function, a byte pointer is declared. In the next one this pointer is made to point to the actual value of the parameter address#. If we call the function in a statement like:

```
PRINT peek(1989)
```

we will get a printout of the contents of the byte at the address 1989. As you see we have made the BASIC function PEEK.

Until now we have primarily used real pointers, but it is possible to create pointer variables pointing to integers, strings, records and arrays. These pointers are declared as shown in the following examples:

```
POINTER t$, n#
POINTER i%, a!
POINTER rec'ptr@ TO freezer'article@
POINTER arr'ptr$ TO string'table$(,,)
```

Note, that in the declaration of pointers to structured variables like records and arrays you must specify more precisely what the pointer should point to. This is because these variables have an inner structure (the number of dimensions in a

table and the field names and types in a record) and in the declaration you specify that the pointer should point to a variable with the same inner structure as the one after the keyword TO.

We have seen how a statement like:

```
^ptr:=<number expression>
```

makes the pointer variable point to a place whose address is the value of the expression on the right side. This expression may be either the address of another variable or it may be the address of another place in the storage (an IO port or may be an address returned by a system routine).

However, it is possible to reserve a special area in the storage as the data field of a pointer variable. This is done by calling the procedure allocate with the pointer variable as a parameter:

```
allocate(ptr)
```

After this call, AmigaCOMAL has reserved an area as a data field and the pointer is pointing to this area.

The data field will be reserved in that area of RAM where the datafields for all other variables are placed. As a consequence, a datafield reserved in a closed procedure will be cleared at the return from the procedure. To prevent this it is possible to specify that the data field should be allocated to a more safe place. This is done at the call to allocate in this way:

```
allocate(ptr,1)
```

As the result of this call AmigaCOMAL places the data field in the storage administered by the Amiga operating system and the content is not cleared at the return from a closed procedure (instead of the parameter 1, another number may be used - see a description of the routine AllocMem from the Amiga operating system).

An area reserved by allocate may be freed by calling the procedure deallocate with the pointer as parameter:

```
deallocate(ptr)
```

Pointer variables are most often used in connection with records to build up lists and trees or they are used in calls to the Amiga operating system.

In the program shown below, ten random numbers are placed in a list and finally printed in increasing order

```
0010 // Demonstration of lists
0020
0030 RECORD list@
0040   POINTER next@ TO list@
0050   FIELD value
0060 ENDRECORD list@
0070 POINTER actual@ TO list@, preceding@ TO list@, new@ TO list@
0090 start#:=0
0100
0110 page
0120 LOOP 10 TIMES
0130   insert(rnd)
0140 ENDLOOP
0150 output
0160
0170 PROC insert(number)
0180   allocate(new@)
0190   new@.value:=number
0200   IF start#=0 THEN
0210     start#:=^new@
0220   ELSE
0230     ^actual@:=start#; ^preceding@:=0
0240     passed:=(number<actual@.value)
0250    WHILE (NOT passed) AND ^actual@<>0 DO
0260       ^preceding@:=^actual@
0270       ^actual@:=^preceding@.next@
0280       IF ^actual@<>0 THEN passed:=(number<actual@.value)
0290    ENDWHILE
0300     IF ^preceding@=0 THEN       // Have to be placed first in list
0310       ^new@.next@:=start#
0320       start#:=^new@
0330     ELIF ^actual@=0 THEN        // Have to be placed at the end
0340       ^preceding@.next@:=^new@
0350     ELSE                        // place in the inner of the list
0360       ^new@.next@:=^preceding@.next@
0370       ^preceding@.next@:=^new@
0380     ENDIF
0390   ENDIF
0400 ENDPROC insert
0410
0420 PROC output
0430   ^actual@:=start#
0440   WHILE ^actual@<>0 DO
0450     PRINT actual@.value
0460     ^actual@:=^actual@.next@
0470   ENDWHILE
0480 ENDPROC output
```

In the program, new data fields are created at the time they are needed. Pointer variables are sometimes called dynamic variables.

In Chapter 6. you will find other ways to use pointers and records.

5. - Packages

A package is a collection of procedures, functions or other variables stored as a disk file. Packages may be written in either AmigaCOMAL (COMAL packages), or in assembler or C (machine coded packages). Regardless of the language used to develop the package, it is used in the same way and the procedures and functions in the package are used in exactly the same way as procedures an functions in a normal program.

Packages may be used to build up an easily accessible library of useful routines or it may be used to divide large programs into smaller parts (modular programming).

5.1 Using Packages

The procedures, functions and other variables in a package are made accessible by the command:

USE *package name* [FROM *directory path*]

where the FROM part in the square brackets may be omitted. This command may be either a direct command or an AmigaCOMAL program statement. Such program statements must be the first executable lines in the program. [This is different than IBM and Commodore 64 COMALs, where the USE statement does not have to be first in the program.]

When a USE command is executed, AmigaCOMAL searches for a package in RAM with the name given as parameter. If the package is not found there, AmigaCOMAL searches for the package on disk.

Example: If you want to use the routines in the Graphics package you execute the command:

USE Graphics

It is possible to get a list of all packages currently in RAM by executing the command:

LISTPACK [[TO] *file name*]

You can get a list of all the routines in a specified package by using the command:

LISTPACK *package name* [[TO] *file name*]

Example: To get a printer listing of all the routines in the graphics package you would type:

LISTPACK *Graphics* TO "lp:"

When a package is read into RAM from disk it will stay in RAM until all packages are removed by the command:

DISCARD

5.2 Programming Packages

A package may be written in AmigaCOMAL (COMAL package) or it may be written in assembler, C or another suitable compiler (machine coded package).

In this section we will describe how to develop COMAL coded packages. Developing machine coded packages will be discussed in the Development Manual.

A COMAL coded package is simply a normal AmigaCOMAL program with some EXPORT statements added at the beginning and then saved to disk with the SAVE command, specifying a file name that ends with .cmp.

Example: The following package contains only one procedure, rand, which is an improved version of the rnd function in AmigaCOMAL.

```
0010  // Random package
0020
0030 EXPORT rand
0040
```

```
0050 x:=1; y:=10000; z:=3000 // Seeds
0060
0070 FUNC rand CLOSED
0080   x:=171*(x MOD 177)-2*(x DIV 177)       // First generator
0090   y:=172*(y MOD 176)-35*(y DIV 176)      // Second generator
0100   z:=170*(z MOD 178)-63*(z DIV 178)      // Third generator
0110   temp:=x/30269.0+y/30307.0+z/30323.0    // Combine
0120   RETURN temp-int(temp)
0130 ENDFUNC rand
```

This example shows a typical COMAL package. The start of the program specifies which procedures, functions or other variables are to be exported. This is done with the EXPORT statement.

The main part of the program works as an initialization part. This part is executed only once and just after the package is read from disk (via the USE command). In the example, the initialization part is used to set the seeds. In many packages you don't need an initialization. Such packages will contain no main part (only procedures and functions). Procedures and functions that are to be exported from a package must be closed.

COMAL packages cannot be stopped via a break. If you break a program while it is inside a package, the break will be suspended until it returns to the main program. The package may test for break using the esc function just as if it had executed a TRAP ESC-.

It is possible for a package to receive messages about changes in state of the AmigaCOMAL system. For instance, when AmigaCOMAL is about to execute a RUN command AmigaCOMAL checks if there is a closed procedure named signal in each active package. If so, a message is sent to the signal procedure. Signals are numbers sent in the following cases:

1: USE of this package
2: DISCARD
3: NEW
4: variables being cleared
5: RUN
6: CON
7: command error / program end by error
8: END by END statement
9: program stop that may be CONtinued / end without END statement
10: BYE

A 5 would be passed to signal if a RUN command was about to be executed. If two or more packages have a signal procedure, the last one USEd is passed the signal first (signal passed in reverse order).

Example: Let us say that we want the same sequence of random number seach time a program is run. Then we have to place a signal routine in the package. This signal routine must set the seeds to the start values each time it receive a RUN signal. The package may now look like this:

```
0010  // Random package
0020
0030 EXPORT rand
0040
0050 x:=1; y:=10000; z:=3000 // Seeds
0060
0070 FUNC rand CLOSED
0080   x:=171*(x MOD 177)-2*(x DIV 177)       // First generator
0090   y:=172*(y MOD 176)-35*(y DIV 176)      // Second generator
0100   z:=170*(z MOD 178)-63*(z DIV 178)      // Third generator
0110   temp:=x/30269.0+y/30307.0+z/30323.0    // Combine
0120   RETURN temp-int(temp)
0130 ENDFUNC rand
0140
0150 PROC signal(s) CLOSED
0160   IF s=5 THEN  // RUN-signal
0170     x:=1; y:=10000; z:=3000              // Seeds
0180   ENDIF
0190 ENDPROC signal
```

A signal routine may be absolutely necessary if the package has called one of the system routines in the Amiga operating system. Let us say you have made a package that opens a window. Then you should have a signal routine that closes this window when it receives either a RUN or a DISCARD signal.

A COMAL package may use other packages (other COMAL packages as well as machine coded packages) and even cyclic USE is allowed. Package P1 may USE package P2 that USEs package P1. A package may EXPORT variables that are imported from another package (re-export). This may be used to expand existing packages.

5.3 Routine Libraries

As mentioned earlier packages may be used to build up an easily accessible library of useful routines. The following package shows an example of this. It

contains a number of mathematical functions that are not standard in AmigaCOMAL.

```
0010  // Function library
0020
0030 EXPORT cot(),inv(),erf()
0040 EXPORT sinh(),cosh(),tanh(),coth()
0050 EXPORT integral(,,)
0060
0070 FUNC cot(x) CLOSED
0080   RETURN cos(x)/sin(x)
0090 ENDFUNC cot
0100
0110 FUNC inv(x) CLOSED
0120   RETURN 1/x
0130 ENDFUNC inv
0140
0150 FUNC sinh(x) CLOSED
0160   u:=exp(x)
0170   RETURN (u-1/u)/2
0180 ENDFUNC sinh
0190
0200 FUNC cosh(x) CLOSED
0210   u:=exp(x)
0220   RETURN (u+1/u)/2
0230 ENDFUNC cosh
0240
0250 FUNC tanh(x) CLOSED
0260   RETURN sinh(x)/cosh(x)
0270 ENDFUNC tanh
0280
0290 FUNC coth(x) CLOSED
0300   RETURN cosh(x)/sinh(x)
0310 ENDFUNC coth
0320
0330 FUNC erf(x) CLOSED              // The normal distributed error function
0340   FUNC g(x) CLOSED
0350     RETURN exp(-x*x/2)
0360   ENDFUNC g
0370   RETURN integral(g(),0,x)/sqr(2*pi)+0.5
0380 ENDFUNC erf
0390
0400 FUNC integral(REF f(),a,b) CLOSED // Simpson integration
0410   n:=32; dx:=(b-a)/n
0420   xi:=a; s:=(f(a)-f(b))*dx/6
0430   LOOP n TIMES xi:=xi+dx; s:=s+(f(xi)+2*f(xi-dx/2))*dx/3
0440   RETURN s
0450 ENDFUNC integral
```

There are many advantages to making such a routine library. Of course you save a lot of typing by simply USEing such a package in the start of a program that needs these functions. But it is also a great advantage that you can change one

or more of the algorithms used in the package without affecting the programs that USE this package.

You could imagine that you would make the function tanh(x) a little bit faster by the following change:

```
FUNC tanh(x) CLOSED
  u:=exp(x)
  RETURN (u-1/u)/(u+1/u)
ENDFUNC tanh
```

and the same for coth(x). These changes will not affect programs that USE the package except that they execute a little bit faster.

A more drastic change would be to write the most time critical of the functions, i.e. the integral function, in machine code. This may still be done without changing the user programs. In the development manual (an extra cost option) we will show how to make such a change.

5.4 Standard Packages

In the Packages drawer on the AmigaCOMAL system disk you will find a collection of standard packages. These are:

```
System
System_code

Exec_library
Dos_library
Intuition_library
Graphics_library
Layers_library
Diskfont_library
Icon_library
Potgo_library
Devices
Messages
Windows
Screens
Gfx
Layers
RastPort
View
```

```
UniGraphics
Turtle

ComalExcept
```

The first two packages make it possible to get information about the Amiga-COMAL system.

The next eight (_library) packages contain interface routines to the libraries in the Amiga operating system.

The following eight packages (Devices, Messages etc.) are all written in AmigaCOMAL. They contain, among other things, definitions of some of the data structures used in the call to the library routines.

The next three packages are graphics packages.

The last package handles exceptions.

The following sections give a short description of these packages.

5.5 System Packages

5.5.1 System_code

This package contains two functions:

FUNC comalstr

Returns the address of the COMAL structure which is an area where AmigaCOMAL stores a number of system information.

FUNC comalwait(signal_mask#)

Like the Exec function Wait, this function waits on the arrival of one or more signals (specified by the signal mask). But contrary to the Exec routine, comalwait() also waits on the signals allocated by the AmigaCOMAL system.

This means that you may break a program that has called comalwait(). The return value of the function is a mask for the signals that has arrived. When you call comalwait with one of intuition-signals in the signalmask, will there in the field cmlintuimessage@ in comal strukture be placed a pointer to a copy of the messages, which intuition has sent. These messages are stored as a list in cmlintuimessage@'s For further information consult a description of the Exec routine Wait.

The contents of these structures are:

```
RECORD cmlintuimessage@
  POINTER nextmsg@ TO cmlintuimessage@
  FIELD class#
  FIELD code%
  FIELD qualifier%
  FIELD iaddress#
  FIELD mousex%, mousey%
  FIELD seconds#, micros#
  FIELD idcmpwindow#
ENDRECORD cmlintuimessage@
RECORD io_struct@
  FIELD screen#
  FIELD screentype%, screendepth%, screenwidth%, screenheight%
  FIELD window#
  FIELD windowdepth%, charno%, lineno%
  FIELD gzzxoff%, gzzyoff%
  FIELD windowwidth%, windowheight%
  FIELD fontid#
  FIELD fontheight%, fontwidth%, fontbase%
  FIELD virtual#
  FIELD cursor!, softstyle!
  FIELD menuhd#, menybytes#
  FIELD privat1#, privat2#, privat3#      // Don't tuch!
  FIELD conmsg#                  // Console input message // Not active_io
  FIELD conreply# // Console reply port    // Not active_io
ENDRECORD io_struct@

RECORD comalstructure@
  FIELD sp_top#, sp_bottom#
  FIELD pck_link#
  FIELD taskid#
  FIELD dosbase#, intbase#, gfxbase#
  FIELD laybase#, fontbase#, iconbase#
  FIELD eventflag!, eventcount!
  FIELD comwinsig!, comkbdsig!
  FIELD excwinsig!, exckbdsig!
  FIELD serialsig!, timersig!
  FIELD verno!                   // Current AmigaCOMAL version number
  FIELD res1!, res2%
   // Call comalwait with comwinsig! or excwinsig! and the next
   // field wil be pointing to a list of cmlintuimessages (the
   // real content of IntuiMessages).
```

```
  POINTER cmlintuimessage@ TO cmlintuimessage@
  POINTER active_io@ TO io_struct@
  POINTER command_io@ TO io_struct@
  POINTER exec_io@ TO io_struct@
ENDRECORD comalstructure@

RECORD node@
  POINTER ln_succ@ TO node@
  POINTER ln_pred@ TO node@
  FIELD ln_pri!, ln_type!
  FIELD ln_name#
ENDRECORD node@

RECORD list@
  POINTER lh_head@ TO node@
  POINTER lh_tail@ TO node@
  POINTER lh_tailpred@ TO node@
  FIELD lh_type!, lh_pad!
ENDRECORD list@
```

5.5.2 System

This package is written in AmigaCOMAL. It USEs System_code, from which it re-exports comalwait(). It contains the following routines:

FUNC hex$(i#) CLOSED

Returns a string representing the hexadecimal value of the integeri#.

FUNC bin$(i#) CLOSED

Returns a string representing the binary value of integer i#.

FUNC c_string$(REF array!()) CLOSED

Converts a null terminated string (common string format in C) in the byte table array!() to an AmigaCOMAL string.

FUNC peek(address#) CLOSED

Returns content of the byte with address given by the integer variable address#

FUNC poke(address#,value!) CLOSED

Writes the value of the byte variable value! into RAM at the address given by the integer variable address#.

In addition to these routines, the System package defines some data structures to be used in low level programming. These are the two AmigaCOMAL structures io_struct@ and comalstruct@ and the two basic structures node@ and list@ from the Amiga operating system.

The contents of these structures are:

```
RECORD io_struct@
  FIELD screen#
  FIELD screentype%, screendepth%, screenwidth%, screenheight%
  FIELD window#
  FIELD windowdepth%, charno%, lineno%
  FIELD gzzxoff%, gzzyoff%
  FIELD windowwidth%, windowheight%
  FIELD fontid#
  FIELD fontheight%, fontwidth%, fontbase%
  FIELD virtual#
  FIELD cursor!, softstyle!
  FIELD menuhd#, menybytes#
  FIELD privat1#, privat2#, privat3#              // Don't tuch!
  FIELD conmsg#        // Console input message   // Not active_io
  FIELD conreply#      // Console reply port      // Not active_io
ENDRECORD io_struct@

RECORD comalstructure@
  FIELD sp_top#, sp_bottom#
  FIELD pck_link#
  FIELD taskid#
  FIELD dosbase#, intbase#, gfxbase#
  FIELD laybase#, fontbase#, iconbase#
  FIELD eventflag!, eventcount!
  FIELD comwinsig!, comkbdsig!
  FIELD excwinsig!, exckbdsig!
  FIELD serialsig!, timersig!
  FIELD verno! // Current AmigaCOMAL version number
  FIELD res1!, res2%
  // Call comalwait with comwinsig! or excwinsig! and the next
  // field wil be pointing to a list of cmlintuimessages
                      (the // real  content of IntuiMessages).
  POINTER cmlintuimessage@ TO cmlintuimessage@
  POINTER active_io@ TO io_struct@
  POINTER command_io@ TO io_struct@
  POINTER exec_io@ TO io_struct@
ENDRECORD comalstructure@
```

```
RECORD cmlintuimessage@
  POINTER nextmsg@ TO cmlintuimessage@
  FIELD class#
  FIELD code%
  FIELD qualifier%
  FIELD iaddress#
  FIELD mousex%, mousey%
  FIELD seconds#, micros#
  FIELD idcmpwindow#
ENDRECORD cmlintuimessage@

RECORD node@
  POINTER ln_succ@ TO node@
  POINTER ln_pred@ TO node@
  FIELD ln_pri!, ln_type!
  FIELD ln_name#
ENDRECORD node@

RECORD list@
  POINTER lh_head@ TO node@
  POINTER lh_tail@ TO node@
  POINTER lh_tailpred@ TO node@
  FIELD lh_type!, lh_pad!
ENDRECORD list@
```

The system package exports io-struct@, node@ and list@ as well as a pointer comalstruct@ to the COMAL structure.

As part of the initialization, the package sets this pointer to point to the content of the COMAL structure by executing the statement:

```
^comalstruct@:=comalstr
```

5.6 Interface Routines to the Amiga Libraries

5.6.1 Library Packages

The following packages contain interface routines to the Amiga libraries:

```
Exec_library
Dos_library
Intuition_library
Graphics_library
Layers_library
Diskfont_library
Icon_library
Potgo_library
IFF_library
```

The names of the routines in the packages are the same as those used to call the routines from a C program except that you always have to use the prefix .cmp (to avoid name conflict). For instance:

```
pckopenlibrary
pcksendio
pckclosescreen
```

The parameters used in the routines are exactly the same as those used in a C program.

Example: To open an IO device like console.device you make the following call in a C program:

```
err=OpenDevice("console.device",0,&ConWrtReq,0)
```

The first parameter is the address of a null terminated string (most C compilers make the actual parameter "console.device" into a pointer to the text). The third parameter is also an address. The second and fourth parameters are 32 bits integers.

In AmigaCOMAL, the corresponding call would look like:

```
name$:="console.device"
err:=pckopendevice(^name$+4,0,^ConWrtReq@,0)
```

Note the first parameter. First you have to make a string variable for the name. The actual parameter is the real content of this variable (the first two words (=four bytes) contain the maximum and actual length of the string variable). AmigaCOMAL always places a null byte after the content of a string variable so the address transferred is in fact an address of a null terminated text.

A complete description of the routines would be beyond the scope of this manual. You can get the information from magazines or one of the books covering this subject.

5.6.2 Support Packages for the Library Routines

In a call to most of the library routines, you have to use one or more data structures. A couple packages contain some of these structures. In addition, some of the packages contain procedures and functions that makes it easier and sometimes safer to call the library routines.

In the following sections we will give a short description of these packages. They are all written in AmigaCOMAL so that they can be examined or even changed.

5.6.3 Messages

Re-exported from

System:

```
comalwait()
```

Exec_library:

```
pckwaitport()
pckfindport()
pckputmsg()
pckgetmsg()
pckreplymsg()
```

Structures:

```
RECORD msgport@
   FIELD mp_node@ OF node@
   FIELD mp_flags!, mp_sigbit!
   FIELD mp_sigtask#
   FIELD mp_msglist@ OF list@
ENDRECORD msgport@
```

Functions and procedures:

FUNC createport(portname$,priority!) CLOSED

Creates a port with the name portname$. The port will be chained into the list of public ports with priority of priority!. A signal is allocated and this signal will automatically be sent at the arrival of a message to the port. The returned value of is the address of the port.

Example:

```
POINTER my'port@ TO msgport@
^my'port@:=createport("MyPort",0)
```

PROC deleteport(port#) CLOSED

Removes a port created by createport().

Example:

```
deleteport(^my'port@)
```

PROC deleteallports CLOSED

Removes all ports created by createport().

FUNC allocsignal(s!) CLOSED

Allocates a signal. The function works like the Exec routine with the same name.

Example:

```
my'signal!:=allocsignal(-1)
```

PROC freesignal(s!) CLOSED

Frees a signal allocated by allocsignal().

Example:

```
freesignal(my'signal!)
```

PROC freeallsignals CLOSED

Frees all signals allocated by allocsignal().

The package contains a signal routine that releases all signals and removes all ports at the receiving of a DISCARD or a BYE signal. This might be dangerous if a message arrives at a later time. But it is also dangerous to let them stay.

Some advice: Clean things up yourself!

5.6.4 Devices

Re-exported from

Exec_library:

```
pckopendevice(,,,)
pckclosedevice()
pckdoio()
pcksendio()
pckcheckio()
pckwaitio()
pckabortio()
```

Structures:

```
RECORD iorequest@
   FIELD mn_node@ OF node@
   POINTER mn_replyport@ TO msgport@
   FIELD mn_length%
   FIELD io_device#
   FIELD io_unit#
   FIELD io_command%
   FIELD io_flags!, io_error!
ENDRECORD iorequest@
```

```
RECORD iostdreq@
   FIELD mn_node@ OF node@
   POINTER mn_replyport@ TO msgport@
   FIELD mn_length%
   FIELD io_device#
   FIELD io_unit#
   FIELD io_command%
   FIELD io_flags!, io_error!
   FIELD io_actual#
   FIELD io_length#
   FIELD io_data#
   FIELD io_offset#
ENDRECORD iostdreq@
```

Commands:

```
cmd_reset%:=1
cmd_read%:=2
cmd_write%:=3
cmd_update%:=4
cmd_clear%:=5
cmd_stop%:=6
cmd_start%:=7
cmd_flush%:=8
```

Functions and procedures:

FUNC createstdio(port#) CLOSED

Creates a standard IO request. port# is the address of a port (may be created by createport() in the Message package) that the pointer nm_replyport@ is set to point to.

The returned value is the address of the request.

Example:

```
POINTER my'port@ TO msgport@
POINTER wr'req@ TO iostdreq@
^my'port@:=createport("MyPort",0)
IF ^my'port@=0 THEN STOP
^wr'req@:=createstdio(^my'port@)
```

PROC deletestdio(ioreq#) CLOSED

Removes a standard IO request created by createstdio().

Example:

```
deletestdio(^wr'req@)
```

5.6.5 Screens

Structures:

```
RECORD screen@
   POINTER nextscreen@ TO screen@
   POINTER firstwindow@ TO window@
   FIELD leftedge%, topedge%
   FIELD width%, height%
   FIELD mousex%, mousey%
   FIELD flags%
   FIELD title#
   FIELD defaulttitle#
   FIELD barheight!, barvborder!, barhborder!
   FIELD menuvborder!, menuhborder!
   FIELD wbortop!, wborleft!, wborright!, wborbottom!
   FIELD font#
   FIELD viewport@ OF viewport@
   FIELD rastport@ OF rastport@
   FIELD bitmap@ OF bitmap@
   FIELD layerinfo@ OF layer_info@
   FIELD firstgadget#
   FIELD detailpen!, blockpen!
   FIELD savecolor0%
   FIELD barlayer#
   FIELD extdata#
   FIELD userdata#
ENDRECORD screen@

RECORD newscreen@
   FIELD leftedge%, topedge%
   FIELD width%, height%, depth%
   FIELD detailpen!, blockpen!
   FIELD viewmodes%
   FIELD type%
   FIELD font#
   FIELD defaulttitle#
   FIELD gadget#
   POINTER bitmap@ TO bitmap@
ENDRECORD newscreen@
```

constants:

```
screentype%:=$0F
wbenchscreen%:=1
customscreen%:=$0F
showtitle%:=$010
beeping%:=$020
custombitmap%:=$040
```

Functions and procedures:

FUNC openscreen(new'screen#) CLOSED

Opens a screen. new'screen# is an address of an initialized new'screen structure. The value returned is the address of the screen structure.

The function works like the Intuition function with the same name.

Example:

```
POINTER my'screen@ TO screen@
^my'screen@:=openscreen(^my'new'screen@)
```

PROC closescreen(screen#)

Closes a screen opened by openscreen().

Example:

```
closescreen(^my'screen@)
```

PROC closeallscreens CLOSED

```
Closes all screens opened by openscreen().
```

5.6.6 Windows

```
Structures:

   RECORD newwindow@
      FIELD leftedge%, topedge%
      FIELD width%, height%
      FIELD detailpen!, blockpen!
      FIELD idcmpflags#
      FIELD flags#
      FIELD firstgadget#
      FIELD checkmard#
```

```
    FIELD title#
    FIELD screen#
    FIELD bitmap#
    FIELD minwidth%, minheight%
    FIELD maxwidth%, maxheight%
    FIELD type%
ENDRECORD newwindow@

RECORD window@
    POINTER nextwindow@ TO window@
    FIELD leftedge%, topedge%
    FIELD width%, height%
    FIELD mousey%, mousex%
    FIELD minwidth%, minheight%
    FIELD maxwidth%, maxheight%
    FIELD flags#
    FIELD menustrip#
    FIELD title#
    FIELD firstrequest#
    FIELD dmrequest#
    FIELD reqcount%
    FIELD wscreen#
    FIELD rport#
    FIELD borderleft!, bordertop!, borderright!, borderbottom!
    FIELD borderrport#
    FIELD firstgadget#
    POINTER parent@ TO window@, desendant@ TO window@
    POINTER pointer%
    FIELD ptrheight!, ptrwidth!, xoffset!, yoffset!
    FIELD idcmpflags#
    POINTER userport@ TO msgport@, windowport@ TO msgport@
    FIELD messagekey#
    FIELD detailpen!, blockpen!
    FIELD checkmark#
    FIELD screentitle#
    FIELD gzzmousex%, gzzmousey%, gzzwidth%, gzzheight%
    FIELD extdata#, userdata#
    FIELD wlayer#
ENDRECORD window@
```

constants:

```
wbenchscreen%:=$0001
customscreen%:=$000F
windowsizing#:=$0001
windowdrag#:=$0002
windowdepth#:=$0004
windowclose#:=$0008
activate#:=$01000
```

Functions and procedures:

FUNC openwindow(new'window#) CLOSED

Opens a window. new'window# is the address of an initialized new'window structure. The value returned is the address of a window structure.

The function works like the Intuition function with the same name.

Example:

```
POINTER my'window@ TO window@
^my'window@:=openwindow(^my'new'window@)
```

PROC closewindow(window#)

Closes a window opened by openwindow().

Example:

```
closewindow(^my'window@)
```

PROC closeallwindows CLOSED

Closes all windows opened by openwindow().

The package closes all windows in case of DISCARD or BYE.

5.6.7 Gfx

Structures:

```
RECORD bitmap@
   FIELD bytesperrow%
   FIELD rows%
   FIELD flags!
   FIELD depth!
   FIELD pad%
   FIELD planes#(8)
ENDRECORD bitmap@
```

```
RECORD rectangle@
   FIELD minx%, miny%
   FIELD maxx%, maxy%
ENDRECORD rectangle@
```

5.6.8 Layers

Structures:

```
RECORD layer_info@
   FIELD top_layer#
   FIELD check_lp#
   FIELD obs#
   FIELD rp_replyport@ OF msgport@
   FIELD lockport@ OF msgport@
   FIELD lock!
   FIELD broadcast!
   FIELD locknast!
   FIELD flags!
   FIELD locker#
   FIELD fatten_count!
   FIELD bytereserved!
   FIELD wordreserved%
   FIELD layerinfo_extra_size%
   FIELD longreserved#
   FIELD layerinfo_extra#
ENDRECORD layer_info@
```

constants:

```
layersimple%:=1
layersmart%:=2
layersuper%:=4
layerbackdrop%:=$40
layerrefresh%:=$080
```

5.6.9 Rastport

Structures:

```
RECORD rastport@
   FIELD layer#
   POINTER bitmap@ TO bitmap@
   POINTER areaptrn%
```

```
    FIELD tmpras#
    FIELD areainfo#
    FIELD gelsinfo#
    FIELD mask!
    FIELD fgpen!, bgpen!, aolpen!
    FIELD drawmode!
    FIELD areaptsz!
    FIELD linpatcnt!
    FIELD dummy!
    FIELD flags%
    FIELD lineptrn%
    FIELD cp_x%, cp_y%
    FIELD minterms!(8)
    FIELD penwidth%, penheight%
    FIELD font#
    FIELD algostyle!
    FIELD txflags!
    FIELD txheight%, txwidth%
    FIELD txbaseline%
    FIELD txspacing%
    FIELD rp_user#
    FIELD wordreserved%(7)
    FIELD longreserved#(2)
    FIELD reserved!(8)
ENDRECORD rastport@
```

constants:

```
jam1%:=0
jam2%:=1
complement%:=2
inversvid%:=4
```

5.6.10 View

Structures:

```
RECORD viewport@
    POINTER viewport@ TO viewport@
    FIELD colormap#
    FIELD dspins#, sprins#, clrins#, ucopins#
    FIELD dwidth%, dheight%
    FIELD dxoffset%, dyoffset%
    FIELD modes%
    FIELD reserved%
    FIELD rasinfo#
ENDRECORD viewport@
```

5.6.11 IntuitionSupport

Functions and procedures:

FUNC accept(requesttext$,positivetext$,negativetext$) CLOSED

This function activates the file requester on the screen. the returned value is TRUE or FALSE depending on which field the user has activated, *positivetext$* or *negativetext$*.

FUNC mousex CLOSED

Returns the present value of the x-coordinate for mouse (relative with in the executing window).

FUNC mousey CLOSED

Returns the present value of the y-coordinate for mouse (relative with in the executing window).

FUNC addmenu(menutext$,itemtext$,key$) CLOSED

Create a new menu with title *menutext$*. The first menu value is named *itemtext$* and short identification letter is *key$* (no idetification is given if *itemtext$* is empty).

The returned value is the identification nummer for the first item in the menu. If the menu can not be created the returned value is -1.

FUNC additem(itemtext$,key$) CLOSED

Create a new item under the last created menu (*addmenu)*. The item name is *itemtext$* and short identification letter is *key$* (unless *itemtext$* is empty).

The returned value is the identification number of the *itemtext$*. If the item can not be created the returned value is -1.

PROC resetmenu CLOSED

Remove all menu's created by the function's *addmenu* and *additem*.

FUNC intuiwait CLOSED

Wait for a prompt to be returned.

Possible values are:

key pressed	(1) - a key has been pressen
mouse buttom pressed	(2) - left mouse button active
menu selected	(3) - a menu has been selected
	(0) - nothing happening

If *intuiwait* has returned one of the first three values, it is possible test further using *key$* or one of the following functions:

FUNC mousebottomx CLOSED

x-position of mouse pointer, when left mouse button activated.

FUNC mousebottomy CLOSED

y-position of mouse pointer, when left mouse button activated.

FUNC menunumber CLOSED

Returns the menu of the chossen menu (-1 if no menu chossen). Menu identification is the same as the identification returned by *addmenu* and *additem*.

5.6.12 Iff

Functiones and procedures:

PROC save_window(name$) CLOSED

Save the contents of the currently active window as an iff-file on disc.

PROC load_window(name$) CLOSED

Load a picture in iff-format into the currently active window.

PROC setcompress(c) CLOSED

If c=TRUE will all calls to *save_window* cause the pictures to be saved to be compressed before saving. Even for small pictures can this option produce noticable savings in file size.

This pacckage re-exports all of the routines in the iff.library. It is necessary that before using this package you are certain tha the iff.library is in the **libs. directory**. IFF.library software is public domain software and delivered as a service with AmigaCOMAL.

5.7 Graphics Packages

Two different graphics packages are supplied. These packages are:

UniGraphics
Turtle

Graphics is the standard graphics packages. UniGraphics contains graphics routines that are compatible with UniComals C64 COMAL for Commodore 64/128 and PC COMAL for IBM PC and compatibles.

Turtle is a little COMAL package that makes it possible to work with turtle graphics.

5.7.1 Graphics

This standard graphics package contains the following procedures and functions:

UniGraphics

UniGraphics contains:

PROC arc(c1,c2,r,start_angle,arc)

The procedure causes an arc to be drawn with its center at (c1,c2) and with a radius r. The arc has the length arc degrees starting with the angle start_angle.

Example:

```
arc(100,100,50,45,90)
arc(x0,y0,r,v,2*v)
```

PROC background(color)

This procedure sets the back ground color to the value of the parameter color.

Example:

```
background(3)       // Set back ground color to red
```

PROC circle(c1,c2,r)

Draws a circle with its center at (c1,c2) and with radius r. The drawn figure will not be a round circle unles a suitable relation between the vertical and horizontal units has been chosen (window procedure).

The current pen position is unchanged.

Example:

```
window(-160,160,-100,100)
circle(0,0,50)      // Draw a "circular" circle.
```

PROC clear

Clears the window inside the current drawing region set by clip.

PROC clearscreen

Clears the window inside the current viewport frame set by the procedure viewport.

PROC clip(xmin,xmax,ymin,ymax)

The procedure clip defines a working region expressed in window coordinates (see window procedure). All drawings (clear, fill, draw, ...) cannot be seen outside this region.

Example:

```
clip(2.5,4.7,-2,0)
clip(0,cx,cy,cy+10)
```

FUNC depth

Function that returns the depth of the graphics window (the number of bitmaps in the window).

If the depth is 2, there are four colors with numbers 0-3. The number of colors can always be calculated as:

```
colornumber:=2^depth
```

PROC draw(x,y)

Draws a line from the current pen position (x0,y0) to the point with the coordinates (x0+x,y0+y).

The new pen position is the point with the coordinates (x0+x,y0+y).

Example:

```
draw(0,100)  // Draws a vertical line of length 101
```

PROC drawto(x,y)

Draws a line from the current pen position (x0,y0) to the point with the coordinates (x,y).

The new pen position is the point with coordinates (x,y).

PROC fill(x,y)

Fills an area with the current pen color. The area to be filled must contain the point (x,y) and be bounded of either a curve in a color different from the back ground color or by the frame set by clip.

The current pen position is unchanged.

Example:

```
pencolor(1)
circle(300,100,50)        // Draw a circle with pen color 1
pencolor(2)
fill(300,100)             // .. and fill with pencolor 2
```

FUNC getcolor(x,y)

Return color of the point with coordinates (x,y).

The current pen position is unchanged.

FUNC getrgb(color_register)

Returns the color of the specified color register. Written as a hexadecimal number the content is of the form

```
$0rgb
```

where r, g and b are the content of the red, green and blue color (four bits for each color).

If the graphics system is not opned the value -1 is returned.

PROC graphicscreen(mode)

Opens the graphics system.

mode=0: execute window is used as graphics window
mode=1: special execute window is opened in execute screen
mode>1: high res interlace graphics with mode bitmaps

This procedure must be called before any of the graphics routines are called. When the graphics system is opned the command window or the graphics window may be braught in front by pressing

<Shift> + <F1>.

FUNC height

Returns the height of the graphics window in pixels

FUNC inq(no#)

The function inq returns certain of the internal variable of the graphics system. The parameter no# must be an integral number in the range 0-33. The meaning of the returned value can be found in the following table:

Number	Information about	Range	Affected by
0	graphics mode	0-4	graphicscreen
1			
2	text background	0-	textbackground
3	text color	0-	pencolor
4			
5	graphics backgruond	0-	background
6	pen color	0-	pencolor
7	graphics text width	8/10	textstyle
8	graphics text height	8/10	textstyle
9	gr. text direction	0	(dir. cannot be changed)
10	graphics text mode	0-1	textstyle
11			
12	pen inside dr. frame	0-1	most drawing procs
13		1	
14		0	
15	wrapping active	0	(wrap not implemented)
16	pen down	1	only changed in Turtle
17	x-position	0-width	most drawing procs
18	y-position	0-height	most drawing procs
19	viewport xmin	0-width	viewport
20	viewport xmax	0-width	viewport
21	viewport ymin	0-height	viewport
22	viewport ymax	0-height	viewport
23	vindow xmin	real no.	window
24	vindow xmax	real no.	window
25	vindow ymin	real no.	window
26	vindow ymax	real no.	window
27		1	
28		0	
29		0	
30	x-ratio	(rem. 1)	window and viewport

```
31      y-ratio              (rem. 2)  window and viewport
32                           0
33                           0
```

remark 1: (wdxmax-wdxmin)/(vpxmax-vpxmin)
remark 2: (wdymax-wdymin)/(vpymax-vpymin)

PROC interiorcolor(color)

Subsequent calls to the procedure polygon uses the parameter color to fill out the interior of the polygon.

Example:

```
interiorcolor(2)            // Fill with black
interiorcolor(-1)           // No filling
```

PROC loadscreen(name$)

The procedure causes an IFF-picture previous stored by AmigaCOMAL (see savescreen procedure) or by another program (like DeLuxe Paint) to be fetched from disk and placed in the graphics window. The procedure only works satisfactory in graphics mode 0,2,3 and 4. In graphics mode 1 the picture is loaded into the whole screen (and not only the window).

The use of this procedure assumes that the Iff.library has been installed in the LIBS: directory. This library is a public domain library supplied with Amiga-COMAL. If Iff.library is not in LIBS: the call is ignored.

Example:

```
loadscreen("MandelBrot.iff")
```

PROC move(x,y)

Moves witout drawing the pen from the current position (x0,y0) to the point with coordinates (x0+x,y0+y).

PROC moveto(x,y)

Moves witout drawing the pen to the point with coordinates (x,y).

PROC noclip

The procedure cancels any previously defined clip working region (set by the procedure clip) and restores the working region to the current viewport (set by the procedure viewport).

PROC outlinecolor(color)

Subsequent calls to the procedure polygon uses the parameter color as the color of the edges of the polygon.

Example:

```
outlinecolor(3)     // Draw polygon with red edges
```

PROC paint(x,y)

The procedure causes a region to be filled in with the current pencolor. The region to be filled must contain the point (x,y) and it must be limited by the pencolor or the viewport drawing area. The current pen position is unchanged.

Example:

```
pencolor(1)
moveto(0,0)
drawto(500,150)          // Draw a white line
pencolor(3)
circle(300,100,50)       // .. a red circle
paint(300,100)           // .. and fill the circle with red color
```

PROC palette(no)

The procedure is used to set the first four colors of the graphics screen. The colors are selected from one of three palettes:

palette	color 0	color 1	color 2	color 3
0	black	green	red	yellow
1	black	magenta	cyan	white

palette(-1) restores the colors to the original values (set by the Preferences program). Note that when in graphics mode 0 and 1 all the AmigaCOMAL windows will change colors. The proceudre is implemented to insure compatibility with UniComals graphics package. Normally the more general procedure setrgb should be used.

FUNC pencolor(color)

Procedure used to set the color of the pen.

FUNC pixelx

The function returns the width of a picture element (a pixel) expressed in the current window units.

Example:

```
draw(2*pixelx,0)
```

FUNC pixely

The function returns the height of a picture element (a pixel) expressed in the current window units.

Example:

```
draw(0,pixely)
```

PROCEDURE plot(x,y)

The procedure places a dot at the point with coordinates (x,y).

The current pen position is unchanged.

PROC plottext(x,y,text$)

The procedure causes text$ to be written in the graphics window starting at the point with coordinates (x,y).

The current pen position is unchanged.

Example:

```
plottext(100,150,"AmigaCOMAL")
```

PROC polygon(corners#, REF x(), REF y())

The procedure causes a polygon with corners# vertices to be drawn. The coordinates of the vertices are stored in the vectors x() and y(). The number of

elements must be at least the value of corners#. The edges (the perimeter) of the polygon is drawn in the color set by the procedure outlinecolor and the color of the inner region is set by the procedure interiorcolor (unless the interiorcolor is set to the value -1 in which case a tranparent polygon will be drawn).

Example:

```
DIM x(3), y(3)
READ x(),y()
interiorcolor(3)
outlinecolor(2)
polygon(3,x(),y())      // Draw a red triangle with black edges
DATA 150,300,450
DATA 20,150,20
```

NOTE: The procedure uses a routine in the Amiga graphics library. It seems as if this procedure has a bug. If the corners of the polygon falls outside the graphics window the system may crash (the well known GURU will visit you!).

PROC printscreen

The procedure transfers the content of the graphics screen to the printer.

PROC savescreen(name$)

The procedure causes the content of the graphics window to be saved on disk in IFF format. The procedure only works satisfactory in graphics mode 0,2,3 and 4. In graphics mode 1 the whole screen (and not only the window) will be saved.

The use of this procedure assumes that the Iff.library has been installed in the LIBS: directory. This library is a public domain library supplied with Amiga-COMAL. If Iff.library is not in LIBS: the call is ignored.

Example:

```
savescreen("MandelBrot.iff")
```

PROC savescreen(name$,mode)

The procedure works as described above but if the parameter mode has the value 1 the window is saved in compressed for. This may reduce the size significantly.

Example:

```
savescreen("Mandelbrot.iff",1)
```

PROC setrgb(color_register,red,green,blue)

This procedure is used to set the amount of red, green and blue in the specified color register. The color_register is an integral number in the range 0-31 (this number is directly asociated to the color number used by the procedures pencolor, backcolor etc.) and the amount of color is an integral number in the range 0-15.

Example:

After the call

```
setrgb(1,15,0,15)
```

the color number 1 (set by pencolor procedure) has the color called magenta.

The colors of all the color registers may be restored to their original value by palette(-1).

PROC textbackground(color)

The procedure is used to set the background color used by the plottext procedure (see also textstyle).

PROC textscreen

Closes the graphics system.

PROC textstyle(style#)

The procedure defines the manner in which the printout from the procedure plottext will appear on the graphics window. The following values may be used for the parameter style#:

0 normal text
1 bold
2 italic
4 under lined
8 inverse
16 only text (no background)

More values can be set by adding the values.

Example:

```
textstyle(1+2)          // Print in bold and italic
```

PROC textstyle(width#,height#,direction#,mode#)

The proceudre is implemented to insure compatibility with UniComals graphics package and only the last parameter is significant. The meaning of this is:

0 print without background
1 print with background (as 16 above)

PROC viewport(xmin,xmax,ymin,ymax)

Defines a region of the graphics window in which all drawings take place. The parameters refer to the physical window (pixel coordinates).

PROC viewport_to_window(vpx#, vpy#, REF wdx, REF wdy)

The procedure maps the viewport coordinates (the physical window coordinates) to the window coordinates (set by the window procedure).

FUNC width

The function returns the width of the graphics window in pixels.

PROC window(xmin,xmax,ymin,ymax)

The procedure defines a coordinate system in the current viewport. The parameter defines new coordinates for the border of the viewport.

PROC window_to_viewport(wdx, wdy, REF vpx#, REF vpy#)

The procedure converts window coordinates to physical window coordinates (pixel coordinates).

FUNC xcor

The function returns the current x-coordinate of the pen.

FUNC ycor

The function returns the current y-coordinate of the pen.

5.7.2 Turtle.

This package extends UniGraphics with relative graphics routines (Turtle graphics). The package is written in AmigaCOMAL and uses all the routines in UniGraphics.

The command graphicscreen executes an implicit window(-160,160,-100,100)

(in graphics mode 0 - smaller widths in other modes) and draws a turtle (a small triangle) in the mittle of the window.

All procedures and functions in UniGraphics are accessible from the Turtle package and besides the following routines are supplied:

PROC arcl(radius,arc_angle)

The procedure causes an arc to be drawn to the left with the parameter radius as the radius of curvature and subtending an angle arc_angle degrees. The starting point is the current position and the starting direction is the current direction.

Both the current pen position and the current direction is changed.

Example:

```
arcl(10,90)
```

PROC arcr(radius,right_arc)

The procedure causes an arc to be drawn to the right with the parameter radius as the radius of curvature and subtending an angle arc_angle degrees. The starting point is the current position and the starting direction is the current direction.

Both the current pen position and the current drawing direction is changed.

PROC back(x)

The procedure moves the pen x units backwards in relation to the current drawing direction.

The current pen position is changed.

PROC forward(x)

The procedure moves the pen x units forward in the current drawing direction.

The current pen position is changed.

FUNC heading

The function returns the value of the current drawing direction. The direction is indicated in degrees with 0 vertically upward and positive to the left.

PROC hideturtle

The procedure causes the drawing pen (the turtle) to disappear from view in the window.

PROC home

The procedure causes the drawing pen (the turtle) to return to position (0,0) and with drawing direction upward.

FUNC inq(no#)

The function inq returns certain of the internal variable of the graphics system. The parameter no# must be an integral number in the range 0-33. The meaning of the returned value can be found in the following table:

Number	Information about	Range	Affected by
0	graphics mode	0-4	graphicscreen
1			
2	text background	0-	textbackground
3	text color	0-	pencolor
4			
5	graphics backgruond	0-	background
6	pen color	0-	pencolor
7	graphics text width	8/10	textstyle
8	graphics text height	8/10	textstyle
9	gr. text direction	0	(dir. cannot be changed)
10	graphics text mode	0-1	textstyle
11	dr. pen visible	0-1	hideturtle/show turtle
12	pen inside dr. frame	0-1	most drawing procs
13		1	
14		0	
15	wrapping active	0	(wrap not implemented)
16	pen down	1	penup/pendown
17	x-position	0-width	most drawing procs
18	y-position	0-height	most drawing procs
19	viewport xmin	0-width	viewport
20	viewport xmax	0-width	viewport
21	viewport ymin	0-height	viewport
22	viewport ymax	0-height	viewport
23	vindow xmin	real no.	window
24	vindow xmax	real no.	window
25	vindow ymin	real no.	window
26	vindow ymax	real no.	window
27	cos(pen direc.)	[-1;1]	arcl,arcr,setheading
28	sin(pen direc.)	[-1;1]	left,right,home
29	size of pen	0-	turtlesize
30	x-ratio	(rem. 1)	window and viewport
31	y-ratio	(rem. 2)	window and viewport
32		0	
33		0	

```
remark 1: (wdxmax-wdxmin)/(vpxmax-vpxmin)
remark 2: (wdymax-wdymin)/(vpymax-vpymin)
```

PROC left(angle)

The procedure causes the drawing pen to be turned angle degrees to the left in relation to the current drawing direction.

PROC pendown

The procedure causes the pen to be lowered. This means that the turtle will draw when it is moved, except for the procedures move and moveto.

PROC penup

The procedure causes the pen to be lifted. After this the pen will not draw when it is moved, except for the procedures draw and drawto.

PROC right(angle)

The procedure causes the drawing pen to be turned angle degrees to the right in relation to the current drawing direction.

PROC setheading(angle)

The procedure causes the drawing direction to be set to angle degrees from the home direction (positive to the left).

PROC setxy(x,y)

The procedure causes the pen to be moved to the point with coordinates (x,y). If the pen is down it acts like drawto, i.e. a line will be drawn. If pen is up it acts as moveto, i.e. no line will be drawn.

PROC showturtle

The procedure causes the drawing pen (the turtle) to appear in the window.

PROC turtlesize(x)

The procedure defines the size of the turtle.

In the Turtle package the following abbrivations may be used:

```
bk      = back
bg      = background
cs      = clearscreen
fd      = forward
ht      = hideturtle
```

```
lt      = left
pc      = pencolor
pd      = pendown
pu      = penup
rt      = right
seth    = setheading
st      = showturtle
textbg  = textbackground
```

Most programs developped to be used with UniGraphics will run without changes with the Turtle package. In some cases it is necesary to set the coordinate system back to the physical pixel coordinate system by using the line:

```
window(0,width-1,0,height-1)
```

5.8 Exceptions in AmigaCOMAL

With this package it is possible to write exception routines (a sort of interrupt routines) in AmigaCOMAL. The package contains the following two procedures:

PROC addexcept(REF except(), signalmask#, resident)

`except()` is a procedure with a signal mask as parameter.

`signalmask#` is a mask for the signals that is to cause exception.

`resident=TRUE` : should be used if the routine is part of a package.

After the call of this routine the procedure except() will be called each time one of the signals in the signal mask has caused an exception. The parameter in this call is a mask for the signals that actually has caused the exception.

PROC remexcept(REF except())

Disables the effect of addexcept.

6. - Program development in AmigaCOMAL

AmigaCOMAL is a modern programming language containing advanced program structures (in the form of loops and branches), procedures and functions that may be called recursively, various data types (in the form of arrays and records) and the capability of modular programming techniques, to build up program libraries (using packages).

In the following sections we give some brief examples of how programs may be developed in AmigaCOMAL.

The first example develops a program that makes a small maze. The second example is a small database program. It is a larger program, so it is reasonable to divide it into smaller parts (modules). These parts are stored as packages that are used by the main program. The final example shows how the system routines in the Amiga operating system can be called from an AmigaCOMAL program.

6.1 A Maze

Before starting this project, sit down and think about how a maze is built. If you do that you will realize that a maze is, seen from outside, a box with an entrance and an exit.

Thus, let us start by drawing a box with holes in two opposite corners. This may be done in this way:

```
moveto(xs+e,ys)          // First make an outer frame with a hole
draw((w-1)*e,0)
draw(0,(h-1)*f)
move(0,f)                // .. in each corner
draw(-e*w,0)
draw(0,-f*h)
```

Here e and f are horizontal and vertical units respectively while w and h is width and length respectively measured in the two units e and f (set in the start of the program). Finally xs and ys are the coordinates of the lower left corner of the box.

Note that we are using short variable names in the examples. However, longer variable names could be used if you wanted. For example, instead of E and F, you could have variables named HORIZONTAL and VERTICAL. A short variable name is nice for a quick example. Long variable names often are better as they make the program listing more readable.

The box made by this little program is in fact a very simple little maze. If it is not complicated enough (and it hardly is!) we may divide it into two boxes and connect the two halves with a hole. Now we have two simple mazes and by combining them we have got a more complicated maze than the first one.

Each of the two mazes may be divided into two and combined with a hole. And each of these may be divided etc. We will continuing in this way until there is nothing to divide. Let us say that this is the case if either height or width is one unit.

From this discussion it is seen that the problem is to make a procedure that divides a box into two and takes care of the next division. The input to this procedure should be a "box" i.e. the coordinates of say the lower left corner (X and Y) and the width and height (W and H).

Such a procedure may look like this:

```
PROC maze(x,y,w,h) CLOSED
  IF w>1 AND h>1 THEN
    moveto(xs+x*e,ys+y*f) // Start in the lower left corner
     IF rnd>h/(h+w) THEN  // divide with a vertical line
      d:=rnd(1,w-1)       // choose a point at random
      move(d*e,0)         // .. and make a vertical line
      hole:=rnd(0,h-1)    // .. with a hole placed at random
      draw(0,hole*f)
      move(0,f)
      draw(0,(h-hole-1)*f)
      maze(x,y,d,h)       // Make a maze in the left half
      maze(x+d,y,w-d,h)   // .. and one in the right
    ELSE                  // divide with a horizontal line
      d:=rnd(1,h-1)       // choose a point at random
      move(0,d*f)         // .. and make a horizontal line
      hole:=rnd(0,w-1)    // .. with a hole placed at random
      draw(hole*e,0)
      move(e,0)
      draw((w-hole-1)*e,0)
      maze(x,y,w,d)       // make a maze in the lower half
      maze(x,y+d,w,h-d)   // .. and one in the upper
    ENDIF
  ENDIF
ENDPROC maze
```

This procedure is made such that the division in the two halves is random and decision whether the division should be with a horizontal or a vertical line is made at random, too. Also the position of the hole is chosen at random. The complete program looks like this:

```
0010  // Maze
0020
0030 USE PCGRAPHICS              // USE must be the first executable line in
0035                             // program
0040
0050 GLOBAL e,f,w,h,xs,ys
0060 graphicscreen(0)
0070 pen:=2
0080 pencolor(pen);backcolor(1)
0090 clear
0110 e:=18; f:=8                 // Cell width and  height
0120 w:=30; h:=20                // Number of cells horizontal and vertical
0130 xs:=20; ys:=10
0140
0150 RANDOMIZE
0160 moveto(xs+e,ys)             // First make an outer frame with a hole
0170 draw((w-1)*e,0)
0180 draw(0,(h-1)*f)
0190 move(0,f)                   // .. in each corner
0200 draw(-e*w,0)
0210 draw(0,-f*h)
0220 maze(0,0,w,h)               // Make a maze in the inner
0230 pencolor(1);backcolor(0)
0240
0250 PROC maze(x,y,w,h) CLOSED
0260   IF w>1 AND h>1 THEN
0270     moveto(xs+x*e,ys+y*f)   // Start in the lower left corner
0280     IF rnd>h/(h+w) THEN     // divide with a vertical line
0290       d:=rnd(1,w-1)         // choose a point at random
0300       move(d*e,0)           // .. and make a vertical line
0310       hole:=rnd(0,h-1)      // .. with a hole placed at random
0320       draw(0,hole*f)
0330       move(0,f)
0340       draw(0,(h-hole-1)*f)
0350       maze(x,y,d,h)         // Make a maze in the left half
0360       maze(x+d,y,w-d,h)     // .. and one in the right
0370     ELSE                    // divide with a horizontal line
0380       d:=rnd(1,h-1)         // choose a point at random
0390       move(0,d*f)           // .. and make a horizontal line
0400       hole:=rnd(0,w-1)      // .. with a hole placed at random
0410       draw(hole*e,0)
0420       move(e,0)
0430       draw((w-hole-1)*e,0)
0440       maze(x,y,w,d)         // make a maze in the lower half
0450       maze(x,y+d,w,h-d)     // .. and one in the upper
0460     ENDIF
0470   ENDIF
0480 ENDPROC maze
```

The mazes made by this program are not particularly complicated. But if it happened that a maze is constructed that is very difficult to get through we should make procedure that can find the route.

Such a procedure should systematicly try all possible routes in the maze until the exit is reached. The problem is to find this systematic method.

Take a look of one of the mazes constructed by our program. Note that it consists of a number of small quadratic cells. To get through the maze we have to go into one of these. The cell we enter is open where we entered it. The three other sides are either open or closed. If a side is open we try entering the neighboring cell. In this way, all possible routes are tested and finally the right one is found.

Thus, let us do the following: in turn, examine the neighboring cells that we can enter. For each cell we enter we will do the same, i.e. we will in turn examine the neighboring cells that we can enter. In this way we will find the exit. The following procedure uses this idea:

```
PROC find_route
  PROC examine(c1,c2,rx,ry) CLOSED
   IMPORT found,pen
    FUNC cell_open
      RETURN pen<>readpixel(xs+c1*e+(1+rx)*e/2,ys+c2*f+(1+ry)*f/2)
    ENDFUNC cell_open
    rx:=-rx; ry:=-ry
    IF c1=0 AND c2=0 THEN
      found:=true
      moveto(xs+e/2,ys-f/2)
      draw(0,f)
    ELSE
      LOOP 3 TIMES
        temp:=rx; rx:=-ry; ry:=temp
        IF cell_open THEN
          examine(c1+rx,c2+ry,rx,ry)
          IF found THEN
            draw(-2*rx*e/2,-2*ry*f/2)
            RETURN
          ENDIF
        ENDIF
      ENDLOOP
    ENDIF
  ENDPROC examine
  found:=false
  pencolor(3)
  examine(w-1,h-1,-1,0)
  draw(e,0)
  pencolor(1)
ENDPROC find_route
```

The method we have used to create the maze is sometimes called "divide and conquer". The idea is that the original problem is divided into two (or more) smaller problems of the same type. After having solved these smaller problems they are combined to form the solution to the original problem.

A lot of problems which at first seem unsolvable may be solved using this method. AmigaCOMAL supports fully the method.

6.2 A Small Database Program

A database may contain many different subjects. It might be a database containing information about your records, a database containing information about the content of your refrigerator or a database containing information about people. But independent of the content it must be possible to do the following basic operations:

- add new information to the database
- remove information from the database
- edit the database
- list all or part of the database, possibly sorted

It would be natural to make a package that can do all this independently of the content of the database. Such a package would serve as a module not only for the program we are going to make in this section but as a general database core to be used in many different connections. It would be a good addition to a package library.

On the AmigaCOMAL distribution disk a package named DataBase contains such a database core. The package exports the following routines:

PROC open'new(filename$,post@,REF id#,filesize)

Creates and opens a new database.

filename$ is the name of the new database. post@ is a model for a record in the database. id# is an identification word to be used in any subsequent calls to other routines. A zero value indicates an error and the database is not created. filesize is the number of records in the file.

PROC open'old(filename$,post@,id#)

Opens an existing database.

Works as open'new but the file must exist and there is no size specification.

PROC close'db(id#)

Closes a database opened by open'new or open'old.

id# is the identification word returned from open'new or open'old.

PROC add'post(id#,post@,REF after(,),REF postid#)

Places a new record into the database.

id# is the identification word returned from open'new or open'old.
post@ is the record to be added.
postid# is an identification of the new record - zero if there was no room for the record.
FUNC after(post1@,post2@) is a function that should return true if post1@ is after post2@ in the ordering of your records.

PROC rem'post(id#,postid#)

Removes a record from the database.

id# is the identification word returned from open'new or open'old. postid# is the identification returned by add'post or send to the procedure list'proc (see below).

PROC list'all(id#,post@,REF list'proc(,,))

Sends all records to the procedure list'proc.

id# is the identification word returned from open'new or open'old.
post@ is a model for a record in the database.
list'proc(post@,postid#,REF continue#) is a procedure to which all records are sent one by one until continue# is FALSE.

It would be too much to describe this package in detail. It suffices to know that it executes the functions described above.

We are also going to use another package from our package library. This package, EditText, contains only one procedure:

PROC edittext(REF text$)

Works much like the INPUT statement, but contrary to this, it writes the content of the variable text$ and you may now edit the content using some of the well known editing facilities. This procedure makes input and editing much easier.

The two packages described (DataBase and EditText) are written in Amiga-COMAL and may be changed if you feel they can be improved.

By using these packages we don't need to worry about how the records are organized or how they are stored on disk and we can focus on the real task: making a user friendly database program.

The database we are going to create is a database of names and it should be possible to do the following:

- Make new name records
- Change existing records
- Remove names from the database
- List all names
- Quit

The program may in broad outline look like

```
REPEAT
   get users choice of job
   execute the job
UNTIL quit
```

The way we have described the program is sometimes called pseudo code. We will gradually change this pseudo code until we get a normal AmigaCOMAL program.

In the part of the program described as "execute the job" we have to find out which of the five jobs the user has asked for and then execute the job. If we denote the job with the first letter in the job description, the "execute the job" part can be implemented by using a CASE structure. Thus we may change the pseudo code to:

```
REPEAT
      get users choice of job
      CASE job OF
      WHEN M
            make
      WHEN C
            change
      WHEN R
            remove
      WHEN L
            list
      WHEN Q
            quit
      ENDCASE
UNTIL job=Q
```

In the part of the program named "get users choice of job" we first should list all possible jobs on the screen and then get a letter from the keyboard (as the users choice).

It turns out that several times we have to get one of few selected letters from the keyboard (here the letters M, C, R, L and Q). Thus it is natural to make a function returning one of the letters. The parameter of such a function can be a string consisting of all the valid letters. Let us call this function keypress$.

Our program now looks like:

```
REPEAT
  page
  PRINT AT 5,10: "Quit"
  PRINT AT 7,10: "Make"
  PRINT AT 9,10: "Change"
  PRINT AT 11,10: "Remove"
  PRINT AT 13,10: "List"
  PRINT AT 17,10: "What is your choice? ",
  job$:=keypress$("QqMmCcRrLl")
```

```
    CASE job$ OF
    WHEN "M","m"
      make
    WHEN "C","c"
      chan
    WHEN "R","r"
      remove
    WHEN "L","l"
      list_db
    WHEN "Q","q"
      quit
    ENDCASE
  UNTIL job$ IN "Qq"
```

The function keypress$ looks like this:

```
FUNC keypress$(valid'chr$) CLOSED
   REPEAT
      c$:=inkey$
   UNTIL c$ IN valid'chr$
   RETURN c$
ENDFUNC keypress$
```

This is the main part of our program. To complete it we have to do some initialization and define the content of the records. Then we have to write the five jobs. This is done by writing five procedures so that the code you see above is the final version.

The initialization part has to do the following:

- use packages
- declare certain variables as GLOBAL
- define records
- create the necessary variables
- open the database

We won't know some of the points in detail until the whole program is written.

The initialization part looks almost like:

```
USE DATABASE
USE EDITTEXT
//
GLOBAL person@,id#
//
RECORD person@
    FIELD surname$ OF 20
    FIELD first_name$ OF 30
```

```
    FIELD street$ OF 20
    FIELD town$ OF 20
    FIELD telf$ OF 15
ENDRECORD person@
//
DIM job$ OF 1, id#
//
filename$:="ram:database.dat"
open'old(filename$,person@,id#)
IF id#=0 THEN open'new(filename$,person@,id#,20)
```

If it turns out to be necessary, it is possible to add more variables in the GLOBAL and DIM statements.

Now let us focus on the five procedures, starting with the procedure quit. It is the most simple one and after that is written we may test the main program. The procedure quit consists of only one statement:

```
PROC quit CLOSED
   close'db(id#)
ENDPROC quit
```

Somebody might say that it is foolish to make this as a procedure. But by doing this the program will be much more readable. We also may expand the QUIT routine later to do other things such as issuing a good bye message.

Now we can perform our first test. Naturally we may only use the letter "Q" (since this is the only "job" we have included so far) but this is in fact sufficient for us to test the structure and the layout of the program. First save the program, then run it for the test:

```
SAVE "test1"
RUN
```

After having successfully finishing this test we can continue writing the remaining four procedures. The content of the first one might be like this:

```
make
   let the user type in the new record
   let the user confirm it
   if ok then put the record into the database
```

The typing is naturally done field by field so that the pseudo code may be written more precisely as:

make
  for each field
      let the user type in the field
  repeat
      make the necessary corrections
  until accepted or rejected
  if accepted then put the record into the database

Now we may start writing the procedure. A possible solution is:

```
PROC make CLOSED
  DIM asw$ OF 1
  page
  PRINT AT 3,10: "Type in new post:"
  work'rec@:=person@
  FOR line:=1 TO 5 DO input'field(work'rec@,line)
  REPEAT
    PRINT AT 17,10: "Create (y/n or number to be changed) ",
      asw$:=keypress$("yYnN12345")
    PRINT asw$
    IF asw$ IN "12345" THEN input'field(work'rec@,ord(asw$)-48)
    UNTIL asw$ IN "yYnN"
  IF asw$ IN "yY" THEN
    add'post(id#,work'rec@,after(,),post'id#)
    IF post'id#=0 THEN
      page
      PRINT AT 10,10: "No room for the record!"
      PRINT AT 13,10: "Press space "
      asw$:=keypress$(" ")
    ENDIF
  ENDIF
ENDPROC make
```

The typing takes place in a procedure input'field. It is called with a record and a field number as parameters. The record is reset in the line just before the FOR loop. The parameter work'rec@ must be declared and made global in the main program.

The insertion of the record in the database is done by calling the package procedure add'post. The function after(,) transferred to this procedure simply returns the result of comparing the two surnames.

The list procedure is very simple. It is done by a call to the package procedure list'all. One of the parameters of this procedure is a procedure that handles the real listing. In this case it is a local procedure list'post which is calling a GLOBAL procedure write'post.

The complete procedure looks like:

```
PROC list_db CLOSED
  PROC list'post(person@,nr#,continue#) CLOSED
    write'post(person@,nr#)
    PRINT
    PRINT
    IF currow>12 THEN
      PRINT AT 18,10: "Press space ",
      WHILE inkey$<>" " DO NULL
      page
      PRINT
      PRINT
    ENDIF
    continue:=true
  ENDPROC list'post
  page
  PRINT
  PRINT
  list'all(id#,person@,list'post(,,))
  IF currow<>2 THEN
   PRINT AT 18,10: "Press space ",
    WHILE inkey$<>" " DO NULL
  ENDIF
ENDPROC list_db
```

Having finished the two procedures make and list_db, an even greater part of the final program may be tested.

Having successfully completed the test, the only remaining procedures are chan and remove. The content of these are:

chan

specify the record to be changed
find this record
make the changes
get a confirmation of the changes
if ok then
    remove the old record
    add the corrected record

remove

specify the record to be removed
find this record
get a confirmation
if ok then remove the record

There are a lot of similarities in these two pseudo codes. Of course we can take advantage of this. We will not go deeper into the transformation of the pseudo code into a AmigaCOMAL program. You can see the result in the following listing of the final program:

```
0010 USE DATABASE
0020 USE EDITTEXT
0030
0040 GLOBAL person@,work'rec@,id#,postno#,found,find'post
0050 GLOBAL input'field(,),keypress$(),after(,),write'post(,)
0070
0080 RECORD person@
0090   FIELD surname$ OF 20
0100   FIELD first_name$ OF 30
0110   FIELD street$ OF 20
0120   FIELD town$ OF 20
0130   FIELD telf$ OF 15
0140 ENDRECORD person@
0150
0160 DIM work'rec@ OF person@, job$ OF 1, id#, postno#, found
0180
0200 open'old("database.dat",person@,id#)
0210 IF id#=0 THEN open'new("database.dat",person@,id#,20)
0220
0230 REPEAT
0240   page
0250   PRINT AT 5,10: "Quit"
0260   PRINT AT 7,10: "Make"
0270   PRINT AT 9,10: "Change"
0280   PRINT AT 11,10: "Remove"
0290   PRINT AT 13,10: "List"
0300   PRINT AT 17,10: "Choose job ",
0310   job$:=keypress$("QqMmCcRrLl")
0320   CASE job$ OF
0330   WHEN "M","m"
0340     make
0350   WHEN "C","c"
0360     chan
0370   WHEN "R","r"
0380     remove
0390   WHEN "L","l"
0400     list_db
0410   WHEN "Q","q"
0420     quit
0430   ENDCASE
0440 UNTIL job$ IN "Qq"
0450
0460  // End of main program
0480 PROC quit CLOSED
0490   close'db(id#)
0500 ENDPROC quit
0510
0520 PROC make CLOSED
```

```
0530    DIM asw$ OF 1
0540    page
0550    PRINT AT 3,10: "Type in new record:"
0560    work'rec@:=person@
0570    FOR line:=1 TO 5 DO input'field(work'rec@,line)
0580    REPEAT
0590      PRINT AT 17,10: "Create (y/n or number to be changed) ",
0600      asw$:=keypress$("yYnN12345")
0610      PRINT asw$
0620  IF asw$ IN "12345" THEN input'field(work'rec@,ord(asw$)-48)
0630    UNTIL asw$ IN "yYnN"
0640    IF asw$ IN "yY" THEN
0650      add'post(id#,work'rec@,after(,),post'id#)
0660      IF post'id#=0 THEN
0670        page
0680        PRINT AT 10,10: "No room for the record!"
0690        PRINT AT 13,10: "Press space "
0700        asw$:=keypress$(" ")
0710      ENDIF
0720    ENDIF
0730  ENDPROC make
0740
0750  PROC find'post CLOSED
0760    PROC seek(person@,postid#,continue#) CLOSED
0770      IF postid#=postno# THEN
0780        page
0790        PRINT
0800        PRINT
0810        write'post(person@,postid#)
0820        PRINT AT 10,10: "This post (y/n) ? ",
0830        asw$:=keypress$("yYnN")
0840        PRINT asw$
0850        IF asw$ IN "yY" THEN
0860          work'rec@:=person@; found:=true; postno#:=postid#
0880        ENDIF
0890        continue#:=false
0900      ELSE
0910        continue#:=true
0920      ENDIF
0930    ENDPROC seek
0950    TRAP
0960      page
0970      INPUT AT 10,10: "Type in number of the record: ": postno#
0980    HANDLER
0990      RETRY
1000    ENDTRAP
1010    found:=false
1020    list'all(id#,person@,seek(,,)) // Find the record
1030  ENDPROC find'post
1040
1050  PROC chan CLOSED
1060    find'post
1070    IF found THEN
1080      page
1090      PRINT AT 3,10: "Change post:"
```

```
1100      FOR line:=1 TO 5 DO input'field(work'rec@,line)
1110      REPEAT
1120        PRINT AT 17,10: "Correct (y/n or number to change) ",
1130        asw$:=keypress$("yYnN12345")
1140        PRINT asw$
1150      IF asw$ IN "12345" THEN input'field(work'rec@,ord(asw$)-48)
1160      UNTIL asw$ IN "yYnN"
1170      IF asw$ IN "yY" THEN
1180        rem'post(id#,postno#)
1190        add'post(id#,work'rec@,after(,),postid#)
1200      ENDIF
1210    ENDIF
1220  ENDPROC chan
1230
1240  PROC remove CLOSED
1250    find'post
1260    IF found THEN rem'post(id#,postno#)
1270  ENDPROC remove
1280
1290  PROC list_db CLOSED
1300    PROC list'post(person@,nr#,continue#) CLOSED
1310      write'post(person@,nr#)
1320      PRINT
1330      PRINT
1340      IF currow>12 THEN
1350        PRINT AT 18,10: "Press space ",
1360        WHILE inkey$<>" " DO NULL
1370        page
1380        PRINT
1390        PRINT
1400      ENDIF
1410      continue:=true
1420    ENDPROC list'post
1430
1440    page
1450    PRINT
1460    PRINT
1470    list'all(id#,person@,list'post(,,))
1480    IF currow<>2 THEN
1490      PRINT AT 18,10: "Press space ",
1500      WHILE inkey$<>" " DO NULL
1510    ENDIF
1520  ENDPROC list_db
1530
1540  PROC input'field(REF person@,line) CLOSED
1550    CASE line OF
1560    WHEN 1
1570      PRINT AT 6,10: "1. Surname:                ",
1580      edittext(person@.surname$)
1590    WHEN 2
1600      PRINT AT 8,10: "2. First name(s):          ",
1610      edittext(person@.first_name$)
1620    WHEN 3
1630      PRINT AT 10,10: "3. Street:                 ",
1640      edittext(person@.street$)
```

```
1650   WHEN 4
1660     PRINT AT 12,10: "4. Postal number and town: ",
1670     edittext(person@.town$)
1680   WHEN 5
1690     PRINT AT 14,10: "5. Telephone number:       ",
1700     edittext(person@.telf$)
1710   OTHERWISE
1720      // Should not happen
1730   ENDCASE
1740 ENDPROC input'field
1750
1760 FUNC keypress$(valid'chr$) CLOSED
1770   REPEAT
1780     c$:=inkey$
1790   UNTIL c$ IN valid'chr$
1800   RETURN c$
1810 ENDFUNC keypress$
1820
1830 FUNC after(post1@,post2@) CLOSED
1840   RETURN post1@.surname$>post2@.surname$
1850 ENDFUNC after
1860
1870 PROC write'post(person@,no#) CLOSED
1880   PRINT AT 0,10: person@.surname$,", ",person@.first_name$,
1890   PRINT AT 0,40: "No.: ",no#
1900   PRINT AT 0,10: person@.street$
1910   PRINT AT 0,10: person@.town$,
1920   PRINT AT 0,40: "Telephone: ",person@.telf$
1930 ENDPROC write'post
```

6.3 Calling System Routines From AmigaCOMAL

The following sections show how system routines in the Amiga operating system may be called from an AmigaCOMAL program.

The system routines in the Amiga operating system are organized in libraries. Examples of these libraries are:

- Exec.library
- Dos.library
- Intuition.library
- ...

Before you can use one or more of the routines in one of these libraries, the library has to be opened. During the opening process the library is loaded from the system disk into RAM (if it is not already there). Having successfully opened the library you may call the routines in this library. Finally, when the work is done, the library must be closed to release the part of RAM occupied by the library (if no other processes in the Amiga are using this library).

By calling system routines from an AmigaCOMAL program, you have to follow the same procedure. The opening of the library is done by using a package containing interface routines to the system routines in the library. On the distribution disk you will find packages corresponding to most of the libraries in the Amiga. The names of these packages are derived from names of the packages, for instance:

Name of the package	Command
Exec_library.pck	USE EXEC_LIBRARY
Dos_library.pck	USE DOS_LIBRARY
Intuition_library.pck	USE INTUITION_LIBRARY

The names of the routines in the packages are the same as those names used by a C program except that they have the prefix pck (to avoid name conflict). For instance

C name	AmigaCOMAL name
OpenDevice	pckopendevice
SendIO	pcksendio
CloseScreen	pckclosescreen

The types of the routines and the number and types of the parameters are always exactly the same as those used in a C program. Most of the routines require as parameter a pointer to one or another data structure and the returned value from a routine is in many cases a pointer to a data structure, too. Before the call to such routines you have to define RECORDs corresponding to the data structures used by the routine. If the routine returns a pointer to a data structure you have to create a pointer to the RECORD.

It would be far beyond the scope of this manual to describe all the routines and the data structures used. There are more than 400 routines and almost as many different data structures. We will only give a short description of the routines used in the following examples.

The inclusion of RECORDs and POINTERs makes AmigaCOMAL very suitable to low level programming. You may soon become engrossed in this and discover your Amiga. Even skilled Amiga programmers will enjoy the easy access to the routines.

But it is necessary to warn you. There is always a risk while doing low level programming. You should always store your programs (and not only on the RAM: disk) before a test. AmigaCOMAL doesn't protect you from errors and it's easy to make such errors!

6.3.1 Sorted Listing of a Directory

As a first example of low level programming in AmigaCOMAL we will make a sorted listing of a directory on a disk (like dir in the CLI).

Most of the disk routines are found in the Dos library. This is the fact with all the routines used in this section. Thus, in the start of our program we have to:

```
USE Dos_library
```

To be able to make a listing of a directory, you must get a "lock" for this directory. This lock is returned by the pcklock routine in the Dos library. The call is like this:

```
catalog$:="df0:"
lock#:=pcklock(^catalog$+4,-2)
```

The routine has two parameters. The first one must be the address of the name of the directory.

Note: The first two words (four bytes) in a string variable contains the maximum and actual length of the variable respectively. Thus we have to add four to the address of the string variable to get the address of the real content.

The second parameter must be either -1 or -2. In this case we specify -2 in which way we are telling the Amiga that we only want to read. A zero returned from pcklock indicates an error (which may be further examined by a call to pckioerror).

The reading of the directory is done by calling pckexamine once (to initialize the reading) and then pckexnext several times. The parameter to both of these routines is the address of a data area of 260 bytes. At the return this area is filled with information about a directory entry, for instance the name of the file or the subdirectory.

Before calling pckexamine and pckexnext we have to create a RECORD with a content corresponding to that used by the routines. This RECORD looks like:

```
RECORD fileinfo@
   FIELD diskkey#              // drive number
   FIELD direntrytype#         // + = directory, - = file
   FIELD filename$(108)
   FIELD rest$(144)            // no interest her
ENDRECORD fileinfo@
```

The last 144 bytes in this RECORD contain, among other things, the length of the file and date of creation. We don't need this information in this program, but the size of the data area must be 260 bytes.

The routines pckexamine and pckexnext require the address of the data area being long word aligned. This may not be the case with our record fileinfo@ so we have to create a pointer to fileinfo@ and then allocate a resident data field for this pointer:

```
POINTER info@ TO fileinfo@
allocate(info@,1)                    // Long word boundary!
```

We can now get an unsorted listing of the directory in this way:

```
status#:=pckexamine(lock#,^info@)
IF status#<>0 THEN
  status#:=pckexnext(lock#,^info@)
  WHILE status#<>0 DO                // Read catalog and display
    PRINT c_string$(info@.filename!())
    status#:=pckexnext(lock#,^info@)
  ENDWHILE
ENDIF
pckunlock(lock#)
```

If the returned value from pckexamine or pckexnext is zero every thing is ok. If not, it is probably because there are no more entries, but there may be other reasons (may be examined by calling pckioerror).

The name of the catalog entrance is stored as a null terminated string (common C format). This string is converted to the AmigaCOMAL string format by calling the function c_string$ in the SYSTEM package.

After the reading we politely return the lock to the AmigaDOS by calling pckunlock (otherwise you can't get rid of the icon of that disk - I think you have seen that problem!).

The goal was to get the names in the directory listed in alphabetic order and if possible with all directories first (as in CLI). To do this we have to collect all the names before we display them sorted.

The sorting can be done in different ways. One way is to store all the names in an array and then sort this array before printing the names. There are two disadvantages using this method. First, we don't know the size of the array. Second, the sorting is very often a time consuming process.

It is far better to use dynamic variables and then place the names in either a list or a tree. The sorting is achieved by placing the names in the correct place in the list or the tree. In this case a tree is the easiest to administrate.

Now what is a binary tree? Well we can make the following recursive definition of this:

> A binary tree is either empty or it consists of a note from which two binary trees grows.

We will place the names in the nodes of the tree in such a way that all the names in the left of the two trees should be displayed before the name in the node and this name should be displayed before the names in the right tree. The nodes of the tree will be represented by a RECORD consisting of a name and two POINTERs to the left and the right tree respectively:

```
RECORD cat'entrance@
   FIELD name$ OF 30
   POINTER left@ TO cat'entrance@
   POINTER right@ TO cat'entrance@
ENDRECORD cat'entrance@
```

Due to the recursive definition it is not surprising that the inserting as well as the printing of the tree is done by using recursive procedures. The insertion procedures look like:

```
PROC insert(REF catalog'ptr@,file'name$)
  IF ^catalog'ptr@=0 THEN                    // Top of tree
    allocate(catalog'ptr@)
    catalog'ptr@.name$:=file'name$
  ELIF catalog'ptr@.name$>file'name$ THEN  // seek to the left
    insert(catalog'ptr@.left@,file'name$)
  ELSE                                       // seek to the right
    insert(catalog'ptr@.right@,file'name$)
  ENDIF
ENDPROC insert
```

and the listing procedure is even more simple:

```
PROC list_tree(REF catalog'ptr@)
  IF ^catalog'ptr@<>0 THEN
    list_tree(catalog'ptr@.left@)
    PRINT catalog'ptr@.name$;
    list_tree(catalog'ptr@.right@)
  ENDIF
ENDPROC list_tree
```

The start or the root of the tree is a pointer. Since all the directories should be listed before the files, we have to use two trees and two pointers to the roots. They are defined in this way

```
POINTER dir'start@ TO cat'entrance@
POINTER file'start@ TO cat'entrance@
```

Now all the parts of the program are made, so we are able to write the complete program. The final result looks like:

```
0010  // Sorted display of directory
0020
0030 USE SYSTEM
0040 USE DOS_LIBRARY
0050
0060 RECORD fileinfo@
0070   FIELD diskkey#                    // drive number
0080   FIELD direntrytype#               // + = directory, - = file
0090   FIELD filename!(108)
0100   FIELD rest!(144)                  // no interest her
0110 ENDRECORD fileinfo@
0120 POINTER info@ TO fileinfo@
0130
0140 RECORD cat'entrance@
0150   FIELD name$ OF 30
0160   POINTER left@ TO cat'entrance@
0170   POINTER right@ TO cat'entrance@
0180 ENDRECORD cat'entrance@
0190 POINTER dir'start@ TO cat'entrance@
0200 POINTER file'start@ TO cat'entranc
0210
0220 allocate(info@,1)                   // Long word boundary!
0230 catalog$:="df2:"                    // For instance
0240
0250 page
0260 lock#:=pcklock(^catalog$+4,-2)
0270 IF lock#>0 THEN
0280   status#:=pckexamine(lock#,^info@)
0290   IF status#<>0 THEN
0300     status#:=pckexnext(lock#,^info@)
0310     WHILE status#<>0 DO             // Read catalog and sort
0320       IF info@.direntrytype#>0 THEN // Directory
0330        insert(dir'start@,c_string$(info@.filename!())+"(Dir)")
0340       ELSE                          // File
0350         insert(file'start@,c_string$(info@.filename!()))
0360       ENDIF
0370       status#:=pckexnext(lock#,^info@)
0380     ENDWHILE
0390      // List
0400     ZONE 80
0410     list_tree(dir'start@)           // .. first catalogs
0420     ZONE 40
```

```
0430      list_tree(file'start@)           // .. and then files
0440      ZONE 0
0450    ENDIF
0460    pckunlock(lock#)
0470  ENDIF
0480  deallocate(info@)
0490
0500  PROC insert(REF catalog'ptr@,file'name$)
0510    IF ^catalog'ptr@=0 THEN  // Top of tree
0520      allocate(catalog'ptr@)
0530      catalog'ptr@.name$:=file'name$
0540    ELIF catalog'ptr@.name$>file'name$ THEN  // seek to the left
0550      insert(catalog'ptr@.left@,file'name$)
0560    ELSE  // seek to the right
0570      insert(catalog'ptr@.right@,file'name$)
0580    ENDIF
0590  ENDPROC insert
0600
0610  PROC list_tree(REF catalog'ptr@)
0620    IF ^catalog'ptr@<>0 THEN
0630      list_tree(catalog'ptr@.left@)
0640      PRINT catalog'ptr@.name$;
0650      list_tree(catalog'ptr@.right@)
0660    ENDIF
0670  ENDPROC list_tree
```

6.3.2 Programming of IO Devices

On the Amiga, input and output are normally handled by special routines called IO devices (IO stands for Input/Output). There are IO devices to perform input and/or output through the printer, the serial port, or the console to mention only a few. Every IO device has a name, for instance:

- printer.device
- serial.device
- parallel.device
- console.device
- ...

To be able to use an IO device you have to open the device. IO is then performed by sending a special data structure called a message to the device. Having sent this message the program may wait for the device to complete the task or it may continue and perform something else in the mean time.

One of the more special IO devices is the narrator device. A text sent to this device is sent out through speakers connected to the Amiga (for instance the speakers in the Amiga 1081/1084 monitors).

In this section we will make a program by which you can make the Amiga pronounce a text.

The first thing to do in a program that is to perform IO is to create the data structure (the message) used by the device. Most of the devices use a standard structure called iostdreq@. This one is defined in the package Message.pck. Some devices use an extended version of this structure. This is also the case with our narrator device.

The narrator structure looks like:

```
RECORD narrat@
   FIELD message@ OF iostdreq@   // Standard IO request
   FIELD rate%                   // speaking rate (words/min)
   FIELD pitch%                  // Baseline pitch (in Hz)
   FIELD mode%                   // Pitch mode (0=with, 1=without)
   FIELD sex%                    // Sex of the voice
   FIELD ch_masks#               // Address of audio mask
   FIELD nm_masks%               // Number of masks
   FIELD volume%                 // Volume (0 .. 64)
   FIELD sampfreq%               // Sampling frequency
   FIELD mouths!                 // ?
   FIELD chanmask!               // Used by the Amiga internally
   FIELD numchan!                // Used by the Amiga internally
   narrat@
ENDRECORD
```

The audio mask specifies which of the four sound channels you are going to use and how you will use it. Without further explanation we state how they are created and initialized:

```
DIM amaps!(4)
amaps!(1):=3; amaps!(2):=5; amaps!(3):=10; amaps!(4):=12
narrat@.ch_masks#:=^amaps!()
narrat@.nm_masks%:=4
```

The other fields in the structure are set to standard values by the device during the opening procedure. Before we can open the device, we have to create a reply

port used by the device to return the message to after completion of the IO. A port may be created by the routine createport() in the Message package:

```
POINTER reply@ TO msgport@
^reply@:=createport("SpeakReply",0)
```

If createport() returns a zero an error has occurred.

Now we have to place the address of the port in our message:

```
^narrat@.message@.mn_replyport@:=^reply@
```

At last we are ready to open the device. This is done by calling the Exec routine pckopendevice with the addresses of the name of the device and the message as two of the four parameters:

```
name$:="narrator.device"
IF pckopendevice(^name$+4,0,^narrat@,0)<>0 THEN GOTO TERMINATE
```

The texts to be send to the narrator device must be written in a special phonetic script. A normal (English) text may be translated to this special phonetic script by the function pcktranslate from the package Translator_library that opens the translator library. This library is the smallest of all the libraries in the Amiga since it contains only this function.

The call to pcktranslate is performed in this way:

```
text$:="Hello world"
dummy:=pcktranslate(^text$+4,len(text$),^trans'out!(),256)
```

The array transout!() is a 256 bytes array used by pcktranslate to store the translated text.

The translated text is immediately sent to the narrator device:

```
narrat@.message@.io_command%:=cmd_write%
narrat@.message@.io_length#:=len(c_string$(trans'out!()))
narrat@.message@.io_data#:=^trans'out!()
dummy:=pckdoio(^narrat@)          // Send a message to device and wait
```

Here we have used the routine pckdoio (from the Exec library) and our program will go to sleep until the narrator has completed its work. This is the simplest way to use an IO device.

After this we must close the device and remove the reply port before the program may terminate. The complete program looks like:

```
0010  // Demonstration of IO programming in AmigaCOMAL
0020
0030 USE SYSTEM
0040 USE MESSAGES
0050 USE DEVICES
0060 USE TRANSLATOR_LIBRARY
0070
0080  // Standard write request for the narrator.device
0090 RECORD narrat@
0100   FIELD message@ OF iostdreq@  // Standard IO request
0110   FIELD rate%                 // speaking rate (words/min)
0120   FIELD pitch%                // Baseline pitch (in Hz)
0130   FIELD mode%                 // Pitch mode (0=with, 1=without)
0140   FIELD sex%                  // Sex of the voice
0150   FIELD ch_masks#             // Address of audio mask
0160   FIELD nm_masks%             // Number of masks
0170   FIELD volume%               // Volume (0 .. 64)
0180   FIELD sampfreq%             // Sampling frequency
0190   FIELD mouths!               // ?
0200   FIELD chanmask!             // Used by the Amiga internally
0210   FIELD numchan!              // Used by the Amiga internally
0220 ENDRECORD narrat@
0230
0240  // Define array for mask to sound channel
0250 DIM amaps!(4)
0260  // Initialize
0270 amaps!(1):=3; amaps!(2):=5; amaps!(3):=10; amaps!(4):=12
0280
0290  // Create output buffer for translate procedure
0300 DIM trans'out!(256)
0310
0320  // Create reply port
0330 POINTER reply@ TO msgport@
0340 ^reply@:=createport("SpeakReply",0)
0350 IF ^reply@=0 THEN GOTO TERMINATE  // Stop if zero
0360
0370  // Initialize narrator message
0380 ^narrat@.message@.mn_replyport@:=^reply@
0390 narrat@.ch_masks#:=^amaps!()
0400 narrat@.nm_masks%:=4
0410
0420  // Open 'narrator.device'
0430 name$:="narrator.device"
0440 IF pckopendevice(^name$+4,0,^narrat@,0)<>0 THEN GOTO TERMINATE
0450
0460  // Ready to go!
0470 text$:="Hello world"           // It is a tradition to say this
0480  // Translate to phonetic script
0490 dummy:=pcktranslate(^text$+4,len(text$),^trans'out!(),256)
0500
0510 narrat@.message@.io_command%:=cmd_write%
```

```
0520 narrat@.message@.io_length#:=len(c_string$(trans'out!()))
0530 narrat@.message@.io_data#:=^trans'out!()
0540 dummy:=pckdoio(^narrat@)        // Send a message to device and wait
0550
0560 pckclosedevice(^narrat@)       // Close device ..
0570 TERMINATE:
0580 deleteport(^reply@)            // .. and remove reply port
```

6.3.3 A Speech Package

On the basis of the program in the preceding section we are now going to make a speech package. This package should export the following procedures/-functions:

```
FUNC translate$(t$)
```
translate the text t$ to phonetic script

```
PROC pronounce(t$)
```
pronounce the phonetic script t$

```
PROC say(t$)
```
a combination of translate and pronounce

The narrator device should be opened in the initialization part of the package and closed when the package is DISCARDed or the BYE signal is received.

We do not want the package to wait for the narrator device to complete the work each time a text is sent. Instead it should return to the caller. This is done by using pcksendio instead of pckdoio.

But now we have to be careful! We are not allowed to use the message sent to the device until it has been returned to us.

The narrator device returns the message to the reply port we created but did not use explicitly in preceding section. When we are going to send text number 2 (3,4,5,...) we have to wait for the arrival of our message to the reply port. This waiting may be done by calling the Exec routine pckwaitport. But there is a drawback in using this. If no answer arrives we will be hanging there forever. We cannot break the program!

Another and better solution is to make the Amiga send a signal each time a message arrives at our port. Instead of waiting for the message to arrive, we can wait for the signal. It doesn't seem much better than waiting for the message. If the message doesn't arrive, then no signal will be sent and we will hang there waiting for signal forever (!?).

The trick is that you can wait for several signals at a time. If we use the special routine comalwait from the system package, we can wait not only for the signal sent at arrival of the message, but also for the signals allocated by the AmigaCOMAL system (among these, one is sent if you press <Amiga> + <S>.

The routine comalwait is used in exactly the same way as the Exec routine Wait, i.e. it is called with a signal mask as parameter (a long integer with those signal bits set that you are waiting for). The returned value is a mask for the signals that has arrived (and this might be one of the AmigaCOMAL signals!).

This was the theory. Now to the practical side. We have to prepare the system such that a signal is sent each time a message arrives at our port. This is easily done. If the port is created by the routine createport in the message package, this is already done. You only have to get the signal number so that we know what to wait for. This may be done in this way:

```
signal'mask#:=2^reply@.mp_sigbit!
```

where we at the same time read the signal number and create the mask that are to be used in comalwait.

Now the remaining part of the package can easily be made. The final result looks like:

```
0010  // Speech package
0020
0030 USE SYSTEM
0040 USE MESSAGES
0050 USE DEVICES
0060 USE TRANSLATOR_LIBRARY
0070
0080 EXPORT translate$(),pronounce(),say()
0090
0100 RECORD narrat@
0110   FIELD message@ OF iostdreq@  // Standard IO request
0120   FIELD rate%                  // speaking rate (words/min)
0130   FIELD pitch%                 // Baseline pitch (in Hz)
0140   FIELD mode%                  // Pitch mode (0=with, 1=without)
0150   FIELD sex%                   // Sex of the voice
```

```
0160   FIELD ch_masks#              // Address of audio mask
0170   FIELD nm_masks%              // Number of masks
0180   FIELD volume%                // Volume (0 .. 64)
0190   FIELD sampfreq%              // Sampling frequency
0200   FIELD mouths!                // ?
0210   FIELD chanmask!              // Used by the Amiga internally
0220   FIELD numchan!               // Used by the Amiga internally
0230 ENDRECORD narrat@
0240
0250  // Define array for mask to sound channel
0260 DIM amaps!(4)
0270 amaps!(1):=3; amaps!(2):=5; amaps!(3):=10; amaps!(4):=12
0280
0290  // Create output buffer for translate procedure
0300 DIM trans'out!(256)
0310
0320  // Create reply port
0330 POINTER reply@ TO msgport@
0340 ^reply@:=createport("SpeakReply",0)
0350 IF ^reply@=0 THEN REPORT (199) // Stop if zero
0360 signal'mask#:=2^reply@.mp_sigbit!
0370
0380  // Initialize narrator message
0390 ^narrat@.message@.mn_replyport@:=^reply@
0400 narrat@.ch_masks#:=^amaps!()
0410 narrat@.nm_masks%:=4
0420
0430  // Open 'narrator.device'
0440 name$:="narrator.device"
0450 IF pckopendevice(^name$+4,0,^narrat@,0)<>0 THEN
0460   deleteport(^reply@)
0470   REPORT (199)
0480 ENDIF
0490
0500 text$:="Hello, I am the amigacoamal speechpackage"
0510 dummy:=pcktranslate(^text$+4,len(text$),^trans'out!(),256)
0520 narrat@.message@.io_command%:=cmd_write%
0530 narrat@.message@.io_length#:=len(c_string$(trans'out!()))
0540 narrat@.message@.io_data#:=^trans'out!()
0550 pcksendio(^narrat@)             // Send message to device
0560
0570 FUNC translate$(text$) CLOSED
0580   dummy:=pcktranslate(^text$+4,len(text$),^trans'out!(),256)
0590   RETURN c_string$(trans'out!())
0600 ENDFUNC translate$
0610
0620 PROC pronounce(text$) CLOSED
0630   REPEAT
0640     mask#:=comalwait(signal'mask#)
0650   UNTIL (mask# BITAND signal'mask#)<>0
0660   dummy:=pckgetmsg(^reply@)
0670   narrat@.message@.io_command%:=cmd_write%
0680   narrat@.message@.io_length#:=len(text$)
0690   narrat@.message@.io_data#:=^text$+4
0700   pcksendio(^narrat@)           // Send message to device
```

```
0710 ENDPROC pronounce
0720
0730 PROC say(text$) CLOSED
0740   pronounce(translate$(text$))
0750 ENDPROC say
0760
0770 PROC signal(s) CLOSED
0780   CASE s OF
0790   WHEN 2,10                    // DISCARD or BYE
0800     REPEAT
0810       mask#:=comalwait(signal'mask#)
0820     UNTIL (mask# BITAND signal'mask#)<>0
0830     dummy:=pckgetmsg(^reply@)
0840     pckclosedevice(^narrat@)
0850     deleteport(^reply@)
0860   OTHERWISE
0870      // No action
0880   ENDCASE
0890 ENDPROC signal
```

6.3.4 Some Closing Remarks

In the previous section we have seen examples of system programming in AmigaCOMAL. On the distribution disk there are other examples (Install and Terminal).

If you would like to go further into this subject it is necessary to get some information about the system routines. This could be from either magazines or from one of the books covering this subject. You should especially look for articles and books describing the system routines seen from a C programmers point of view. With a minimal knowledge of C programming it is easy to translate the listing of a C program into an AmigaCOMAL program.

Happy programming! And once more: remember to store your programs before testing!

Appendix A.

The AmigaCOMAL disk.

The AmigaCOMAL system disk contains the following files and directories:

AmigaCOMAL	The COMAL interpreter.
S (dir)	Contains the program used to install AmigaCOMAL.
Programs (dir)	Empty.
Packages (dir)	All the packages of chapter 5. are found in this directory as well as a couple of other packages.
Packdev (dir)	This directory contains a number of ReadMe files and two under directorys containing C and Assebler information and source programs.
Iff-Pictures (dir)	This directory contains pictures in Iff format.
LstFiles (dir)	This directory contains many small programs LISTed to disk in ASCII format. Each COMAL keyword has a sample program in this directory showing its use. Use the ENTER command to retrieve a program into COMAL. Or, since the files are in ASCII, you can read them with any file reader.

Appendix B.

The file system in AmigaCOMAL.

The user may communicate with almost any of the external units of the Amiga such as floppy disk, hard disk, ram disk, serial port or printer using the AmigaCOMAL file system.

Three types of files are supported by AmigaCOMAL: program files, sequential data files and random access data files.

B.1 Filenames

The name of a file may be either the name of one of the standard units in AmigaCOMAL or it may be the name of an AmigaDOS file.

The standard units are:

```
"ds:"  the data screen
"kb:"  the keyboard
"sp:"  the serial port
"lp:"  the printer (alternative use "par:")
```

Note that "lp:" uses the printer.device of the Amiga operating system. This means that codes are converted from the standard ANSI printer codes to the codes of your printer. If you do not want this conversion, the AmigaDOS file "PAR:" should be used.

The name of a disk file should follow AmigaDOS file name rules. This means that there are few restrictions.

If there are no periods in the file name, AmigaCOMAL will add the following file types to the name:

```
.lst   ENTER, MERGE, LIST, DISPLAY
.sav   LOAD, SAVE, RUN, CHAIN
.cmp   USE (Comal)
.pck   USE (Code)
.ext   external procedures or functions
```

If AmigaCOMAL is going to create a file which already exists, the suffix .backup is added to the original file first.

B.2 Program Files

Program files may be stored on a disk either as a memory image file or as an ASCII text file, or it may be output to one of the standard output units ("ds:", "sp:" and "lp:") as an ASCII text file.

To store a program as a memory image file (also called a SAVE file) the SAVE command is used. To read back a memory image file the commands LOAD, RUN or CHAIN are used.

If the commands LOAD or SAVE are used without a file name or if the Project menu is used a file name requester is opened. In this requester the path and the file name may be entered. The path and name of the latest LOADed file is suggested by the requester.

To store a program as an ASCII text file the commands LIST or DISPLAY must be used. LIST and DISPLAY stores the file on disk in exactly the same format as the one you see on the screen if LIST and DISPLAY are used without file name. This means that lines are indented and LIST stores the program with line numbers and DISPLAY without line numbers.

ASCII text program files may be edited using a text editor like ED or MEmacs.

ASCII text program files may be retrieved using either the ENTER or MERGE command. If the ENTER command is used the program lines must contain line numbers.

If the ASCII text program file is edited using a text editor or it is created by another COMAL, there might be syntax errors in some of the lines. When a syntax error is encountered, the line is displayed on the screen for you to edit. After editing press <Enter> and COMAL continues reading the file.

It is possible to store and enter program files using the SELECT command. But since SELECT normally affects a running program it is necessary to turn off the Execute Window by using the RUNWINDOW- command. To store a program using SELECT the following command sequence could be used:

```
RUNWINDOW-
SELECT OUTPUT «file name»
LIST
SELECT OUTPUT "ds:"
RUNWINDOW+
```

and to read a text program file use:

```
RUNWINDOW-
SELECT INPUT «file name»
RUNWINDOW+
```

During the input, the lines are displayed on the screen. At the end of the reading an implicit SELECT INPUT "ds:" is executed.

The reading of a file may be stopped at any time by pressing «Esc» or «Amiga» + «S».

B.3 Sequential Data Files

Before using a sequential file, it has to be opened using the OPEN or SELECT statement. Each file opened by the OPEN statement has a file number (some times called a stream number) attached. SELECT is used to redirect:

- output from PRINT to a file
- input to INPUT from a file

When a file is opened by the OPEN statement you have to specify the name of the file, the file number you want to use to access the file and the way you want to use the file (read, write or both). The general format of the OPEN statement (for sequential files) is:

```
OPEN FILE «file number»,«file name»,«mode»
```

The «file number» is an integer in the range 1-32767 and «mode» is one of the following keywords:

```
READ
WRITE
READWRITE
APPEND
```

The READWRITE mode should be used if special devices like "sp:" (the serial port) are to be used for both input and output.

Once opened, you may write to the file (WRITE, READWRITE and APPEND modes) using WRITE FILE or PRINT FILE statements and read from the file (READ and READWRITE modes) using READ FILE or INPUT FILE statements.

The WRITE FILE statement outputs the data in binary format and it is only possible to read such data using the READ FILE statement. The PRINT FILE and the INPUT FILE statements are used in connection with ASCII text files.

After use, the file has to be closed using the CLOSE statement. The general format of a CLOSE statement is:

```
CLOSE [FILE «file number»]
```

Without the file number, all open files will be closed. When a file is closed, the file number is available for use with future file access.

Example: To transfer a COMAL program from another machine (IBM for instance) through the serial port the following program may be used:

```
0010 inputfile$:="sp:"
0020 outputfile$:="ram:temp.lst"
0030
0040 OPEN FILE 1,inputfile$,READ
0050 OPEN FILE 2,outputfile$,WRITE
0060
0070 WHILE NOT eof(1) DO
0080   INPUT FILE 1: line$
0090   PRINT FILE 2: line$
0100 ENDWHILE
0110
0120 CLOSE
```

After the transfer the program may be retrieved by using the command:

```
ENTER "ram:temp.lst"
```

The SELECT command is used to redirect the standard IO streams of AmigaCOMAL (INPUT and PRINT) to a disk file or one of the standard devices. The general format is:

```
SELECT INPUT «file name»      // Redirect input
SELECT OUTPUT «file name»     // Redirect output
SELECT INOUT «file name»      // Redirect input and output
```

The result of executing the SELECT command is that the old input and/or output file (device) is closed and the new file (device) is opened.

B.4 Random Access Files

With a sequential file, the data is read in the same order as it was written to the file. However, with a random access file, data may be read in any order; independent of the order it was written. Only disk files may used as random access files.

A random access file must be created before it is used. This is done by using the CREATE statement where the length and internal structure of the file is specified. The general format of CREATE is:

```
CREATE «file name»,«number of records»,«record length»
```

If the file is going to hold different contents of an AmigaCOMAL RECORD, the value of «record length» may be calculated by using the varsize function. In other cases you have to know the length of each of the fields in the record. Here are some guide lines:

```
string:                 length of string (len function) + 2
floating point:     8 bytes
long integer:       4 bytes
short integer:      2 bytes
byte integer:       1 byte
```

An existing random access file may be opened by using the OPEN statement. The general format of OPEN (random access files) is:

```
OPEN FILE «file number»,«file name»,RANDOM «record length»
```

Normally «record length» is the value used in the CREATE statement. Having opened the file you may write to and read from the file using the WRITE and READ statements or the PRINT FILE and INPUT FILE
statements (ASCII only). The general format of the READ and WRITE statements are in this case:

```
WRITE FILE «file num»,«rec num»[,«byte offset»]: «expr. list»
READ FILE «file num»,«rec num»[,«byte offset»]: «var. list»

PRINT FILE «file num»,«rec num»[,«byte offset»]: «expr. list»
INPUT FILE «file num»,«rec num»[,«byte offset»]: «var. list»
```

«rec number» is an integer in the range 1 - 2147483647. Without the «byte offset» specification the record with the number specified is read from the start, otherwise it is read from the byte number given.

Although the expression list may be any list of expressions, each expression separated by a comma, it is recommended that only variables are used in the list. In this way it is easier to make sure that the same data types are read from the record as was written to the record. AmigaCOMAL has no way to test for correct types.

After use, the file must be closed just as a sequential file.

Appendix C.

Expressions in AmigaCOMAL.

AmigaCOMAL accepts constants, variables and function names in an expression. Expressions are divided into two categories: numeric expressions and string expressions.

C.1 Numeric Expressions

A numeric expression is an expression whose value is a number. The expression may be built by integer or floating point constants, variables and functions separated by the numerical operators. These operators are divided into two categories:

Algebraic Operators:

^	power
*	multiplication
/	division
MOD	modulo
DIV	integer division
+	addition
-	subtraction and monadic minus
BITAND	binary AND
BITOR	binary OR
BITXOR	binary XOR (exclusive or)

Logical Operators:

<	less than
<=	less than or equal
=	equal
>=	greater than or equal
>	greater than
<>	unequal
NOT	negation
AND	logical AND
OR	logical OR
IN	substring in string

The operators are listed in the order of their priority. The priority may be changed by using parentheses as known from basic calculus.

The value of a purely logical expression is either 1 representing the logical value TRUE or 0 representing the logical value FALSE. An exception is the operator IN. For this operator true value is a positive integer representing the number of the first character of the first occurrence of the substring.

Numerical constants may be either decimal, hexadecimal or binary constants. Hexadecimal and binary constants are always integers. Hexadecimal constants are preceded by a dollar sign ($) and binary by a percent sign (%).

Examples: The following expressions are valid numerical expressions in AmigaCOMAL:

```
12                  a simple decimal integer constant (12)
3*4                 algebraic expression (12)
24/(4-2)*2^3        algebraic expression (96)
pi>8                logical expression (0)
2*("e" IN "Len")    mixed numerical and logical (4)
```

C.2 String Expressions

A string expression is an expression whose value is a string. The expressions are built by constants, variables and functions separated by the string expression operators. These operators are:

```
(:)    string selector
 *     multiplication
 +     addition
```

The operators are listed in the order of their priority.

The operator (:) is a rather special operator (purists will not call it an operator at all). It is a monadic operator to be placed after the operand (in contradiction to the operators NOT and monadic minus that are placed in front of the operator). The multiplication operator is asymmetric since the left operand must be a numerical expression and the right operand a string expression.

Examples: The following expressions are valid string expressions in Amiga-COMAL:

```
"Borge R."+" Christensen"     Borge R. Christensen
10*"-"                        ----------

"AmigaCOMAL"(6:10)            COMAL

5*"AmigaCOMAL"(6:8)+"AL"      COMCOMCOMCOMCOMAL

filename$="test"
filename$+".lst"              test.lst
filename$(2:3)                es
```

Appendix D.

Screen and screen control codes.

D.1 The Screen

The AmigaCOMAL windows are built on the following structure:

```
RECORD io_struct@
   FIELD screen#
   FIELD screentype%, screendepth%, screenwidth%, screenheight%
   FIELD window#
   FIELD windowdepth%, charno%, lineno%
   FIELD gzzxoff%, gzzyoff%
   FIELD windowwidth%, windowheight%
   FIELD fontid#, fontheight%, fontwidth%, fontbase%
   FIELD virtual#
   FIELD cursor!, softstyle!
   FIELD menuhd#
   FIELD menubytes#
ENDRECORD io_struct@
```

The following briefly describes the content of io_struct@. To understand the description you have to be familiar with the Amiga system structure (especially the Intuition structures).

screen#

Pointer to the Intuition screen structure. This pointer is pointing to the correct structure even if the window uses the Workbench screen.

screentype%

The value is either $0001 or $000F corresponding to Workbench screen or a custom screen.

screenwidth%

The width of the screen (in pixels).

screenheight%
The height of the screen (in pixels).

window#
Pointer to an Intuition window structure.

windowdepth%
Number of bit planes in the window.

charno%
The width of the window (in characters).

lineno%
The height of the window (in lines).

gzzxoff%
The x-coordinate of the upper left corner of the visible part of the window.

gzzyoff%
The y-coordinate of the upper left corner of the visible part of the window.

windowwidth%
The width of the window (in pixels).

screenheight%
The height of the window (in pixels).

fontid#
Identification of the font used in the window.

fontheight%
Height of the characters in the font (in pixels).

fontwidth%
Width of the characters in the font (in pixels).

fontbase%
Base line of the font.

virtual#
Address of a structure for the virtual window containing the content of the window in normal characters. The structure looks like:

```
RECORD virt@
   FIELD lineoffset% // line offset of cursor
   FIELD charoffset% // char offset in line
   FIELD content!(charno%,lineno%)
ENDRECORD virt@
```

cursor!
Bit 0 = 1 if cursor is on.

softstyle!
Soft style of text (italic, bold, underline).

menuhd#
Pointer to an Intuition menu structure. It is possible to create other menus. To do this allocate memory for the new menu and deallocate the memory for this one.

menubytes!
Number of bytes in the menu structure.

D.2 Screen Control Codes / CHR$(x) Results

ctrl	dec.	hex.	symbol	description
	00	00	NUL	
A	01	01	SOH	Delete to end of line
B	02	02	STX	Delete to end of screen
C	03	03	ETX	Cursor on
D	04	04	EOT	Cursor off
E	05	05	ENQ	
F	06	06	ACK	
G	07	07	BEL	Flash screen
H	08	08	BS	Destructive back space
I	09	09	HT	Right tab
J	10	0A	LF	Line feed
K	11	0B	VT	Left tab
L	12	0C	FF	Clear screen and move cursor home
M	13	0D	CR	Carriage return
N	14	0E	SO	Delete line and scroll up
O	15	0F	SI	Insert line and scroll down
P	16	10	DLE	Inverse on/off
Q	17	11	DC1	Extra half bright
R	18	12	DC2	Underline on/off
S	19	13	DC3	Italic on/off
T	20	14	DC4	Bold on/off
U	21	15	NAK	Move cursor to start of line
V	22	16	SYN	Move cursor to last character of line
W	23	17	ETB	Move cursor to top line
X	24	18	CAN	Move cursor to last line
Y	25	19	EM	Delete character
Z	26	2A	SUB	
	27	2B	ESC	
	28	2C	FS	Move cursor one character right
	29	2D	GS	Move cursor one character left
	30	2E	RS	Move cursor one line up
	31	2F	US	Move cursor one line down

Pen Color

Choose one of 32 colors by printing a chr$(x) where x is a value between 128-159 (hex $80-$9f). Example:

```
PRINT chr$($80+3)   // sets color to 3
```

Print a chr$(17) before and after the color code to shift to the next 32 colors in Extra Half Bright mode (6 bit planes with 64 colors, some early Amiga models may not have this mode). Example:

```
PRINT chr$(17),chr$(130),chr$(17)   // color to 35
```

Background Color

Print a chr$(16) before and after the color code to set the background to the specified color.

Example:

```
PRINT chr$(16),chr$(130),chr$(16)   // background 3
```

NOTE: You must have installed AmigaCOMAL with enough bit planes for the screen to see all the colors. It takes 6 bit planes to display all 64 colors in Extra Half Bright mode.

Appendix E.

Error numbers and error texts.

E.1 Syntax Errors

Syntax errors detected by AmigaCOMAL occur during the examination of a line (a command or a program line). The error is printed in a small Error Window and the cursor is placed where AmigaCOMAL detected the error.

```
0001:       Illegal character

0002:       Illegal line number
            A line number must be in the rage 1-9999.

0003:       String too long
            The maximum length of a text constant is 127.

0004:       Variable expected

0005:       Constant expected

0006:       Number expected

0007:       String expected

0008:       ( expected

0009:       ) expected

0010:       ) without (

0011:       Missing operand or illegal type
            An operator is found but the right operand is missing or is of
            wrong type. A reserved word used as a name may cause this error.

0012:       Illegal operator
            A special character is found at the place of an operator.
            (example:  PRINT 2# )
```

0013: Illegal type
A string expression is used instead of a number or the reverse.

0014: " expected

0015: := expected

0016: , expected

0017: ; expected

0018: : expected

0019: Reserved word
A reserved word (like AT, TO, FOR etc.) is used as a name.

0020: TO expected

0021: DO expected

0022: OF expected

0023: THEN expected

0024: WHEN expected

0025: WHEN illegal in IF..EXIT

0026: FILE expected

0027: Type of OPEN expected
You have to specify how the file is to be used by appending the clauses READ, WRITE, RANDOM, APPEND or READWRITE.

0028: FROM expected

0029: INPUT/OUTPUT expected

0030: Illegal in single IF
Only simple (single line) statements may be used in the single line IF statement.

0032: Illegal in single loops
Only simple (single line) statements may be used in the single line loops.

0044: Not program statement
Commands like LIST, SCAN, RUN may not be used in program lines.

0045: Not direct command
Statements like REPEAT, PROC, DATA may not be used as direct commands.

0055: TIMES expected

E.2 Pre-pass Errors

These errors are detected during the pre-pass scan of the program (SCAN or RUN commands).

0031: Illegal in IF-structure
Statements like DATA, PROC and FUNC may not be used inside an IF structure (and all other structured statements).

0033: Illegal in loops
Statements like DATA, PROC and FUNC may not be used inside a loop structure (and all other structured statements).

0034: WHEN-statement expected
In a CASE structure there must be at least one WHEN part.

0035: Illegal in WHEN-part
Statements like DATA, PROC and FUNC may not be used inside a CASE structure (and all other structured statements).

0036: Illegal in RECORD-structure
The only legal statements are comments, FIELD, RECORD, ENDRECORD and POINTER.

0037: Illegal in TRAP

0038: Illegal in main program
The statements RETURN, IMPORT etc. are illegal in the main program.

0039: Illegal in PROCedure

0040: Illegal in string FUNC

0041: Illegal in real FUNC

0042: Illegal in open PROC/FUNC
The statement IMPORT is illegal in an open PROCedure or FUNCtion.

0046: Label not found
The label is not found within the scope of a GOTO or RESTORE statement.

0047: Two labels
Two (or more) labels with the same name are found within the scope of a GOTO or RESTORE statement.

0048: ENDFOR expected

0049: Wrong name
If a name is specified in ENDFOR, ENDPROC, ENDFUNC and ENDRECORD it must be the same as in the corresponding FOR, PROC, FUNC or RECORD.

0050: UNTIL expected

0051: ENDWHILE expected

0052: ENDLOOP expected

0053: EXIT only in loops

0054: Error in program structure

0056: ENDIF expected

0057: ENDCASE expected

0058: ENDPROC expected

0059: ENDFUNC expected

0060: HANDLER expected

0061: ENDTRAP expected

0082: ENDRECORD expected

E.3 Execution Errors

These errors are found during the execution of a command.

0062: No WHEN in CASE
The calculated CASE value is not fund in one of the WHEN statements and there is no OTHERWISE part.

0063: Out of memory
This error is most probably caused by a DIM statement making a large array.

0064: Line number not found

0065: No program

0066: Not SAVE-file
You have tried to load a file which is not a SAVE file (stored by a SAVE command) or which is stored by an earlier version of AmigaCOMAL. If it is a SAVE file from an earlier version of AmigaCOMAL, load the

program into the earlier version, LIST the program and ENTER it into the new version.

0067: Renumber error
You are trying to renumber outside the line number range 1-9999

0068: CONtinue not allowed
A program execution can only be CONtinued if it was stopped by a STOP statement or by break («Amiga»+«S»). If the program execution stopped because of an error or if was changed since it was stopped it cannot be CONtinued.

0069: IMPORT not allowed
IMPORT is not allowed in EXTERNAL and package procedures or functions.

0070: Inside package - only LIST, MAIN and DISCARD allowed
The program has stopped inside a package. Use the MAIN command to return to the main program.

0071: Not inside HANDLER part

0073: STEP not allowed
STEP is another form of CONtinue. See error number 68.

0075: Wrong no. of parameters

0076: Wrong type of parameter

0077: Unknown identifier

0078: Type error

0079: Double defined variable

0080: Next line not a DATA
You have RESTOREd to a line which is not followed by a DATA statement.

0081: Illegal device or file name
AmigaDOS allows almost any character in a file name so the error is not likely to occur.

0083: Data address not defined
No data area is allocated to a POINTER variable.

0084: Unknown record field

0085: Address error
A POINTER variable must point to an even address (except a byte pointer).

0099: System error
You are lucky you got an error message at all!

0100: Division by zero

0101: Overflow

0102: Argument error

0103: Integer overflow

0104: Missing digit

0105: Conversion error
An error has occurred during the conversion of a binary number to ASCII. This error is most probably caused by a bug in AmigaCOMAL!

0106: Stack overflow
Recursive procedure calls have reached a very deep level

0107: Out of range
This is most probably because you have used an array index outside the range of this dimension.

0108: String too long
The maximum length of a string is 32767.

0109: USING-error
The format string is incorrect.

0110: Undefined function value
The program has reached ENDFUNC in a function. A RETURN statement must be executed prior to this.

0111: Out of DATA
There is no more DATA to READ. The function EOD may be used to test if there is more DATA to READ.

0112: Error in external call

0113: No more SCB's
Too many open streams. Use the Install program to increase the maximum number of open streams.

0114: Stream already open

0115: Stream not open

0116: Illegal read/write mode
You are trying to open a stream (file) in WRITE mode which can only be opened in READ mode (for instance "kb:") or the reverse (for instance "lp:").

0117: End of record
You have reached the end of a record in a file.

0118: EXPORTed identifier undefined
A name in the EXPORT list of a package is unknown.

0119: EXPORTed PROC/FUNC not CLOSED
Only CLOSED functions or procedures may be EXPORTed from a package.

0199: Package initialization error
An error has occurred during the initialization of a package. It is up to the package programmer to inform you of the precise reason.

E.4 AmigaDOS Errors

If an error is returned to AmigaCOMAL by an AmigaDOS routine the number 1000 is added to the AmigaDOS error code. Consult your Amiga manual for details.

1103: Insufficient free store

1202: Object in use

1203: Object already exists

1205: Object not found

1212: Object not of required type

1214: Disk write-protected

1215: Rename across devices attempted

1216: Directory not empty

1218: Device not mounted

1219: Seek error

1221: Disk full

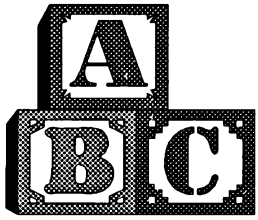
1222: File is protected from deletion

1223: File is protected from writing

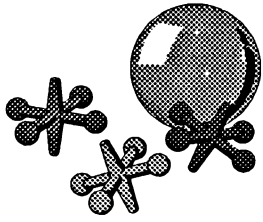
1225: Not a DOS disk

1226: No disk in drive

1232: No more entries in directory



TUTORIAL



REFERENCE

COMPILER

COMPILER

Compiler Module

UniComal

Developers Description

AmigaCOMAL

Copyright Notice

UniComal A/S
Tværmarksvej 19
DK-2860 Søborg
DENMARK

Single CPU License

The price paid for one AmigaCOMAL, including a manual and a diskette, licenses you to use the product on one CPU when and only when you have signed and returned the **License Agreement** printed on the last page of the UniComal Reference Manual.

Disclaimer

UniComal A/S has made every effort to supply a dependable product of the highest possible quality. However, UniComal A/S makes no warranties as to the contents of this manual and the system and supplementary diskettes and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. UniComal A/S further reserves the right to make changes to the specifications of the ??? Module and the UniComal system and the contents of the manuals without obligation to notify any person or organization of such changes. Nevertheless, it is the intention of UniComal A/S to provide all registered users with a periodic newsletter as required providing update information at no charge for a period of one year from the purchase date.

Compiler Module - version 2.10 -

This document was prepared using WordPerfect ver. 5.1
and a HP LaserJet Series II.

Table of Contents

1. Making Stand Alone AmigaCOMAL Programs. 5
1.1 Introduction. 5
1.2 The AmigaCOMAL compiler ComalComp. 5
1.3 Combined compilation and linking using the Compile. 8
1.4 Compiling from WorkBench. 9

2. Package Development. 11
2.1 Introduction. 11
2.2 An example. 12

3. Internal format of data in AmigaCOMAL. 17
3.1 Number format in AmigaCOMAL. 17
3.2 Registers and use of storage. 18
3.3 Transferring of parameters. 18
3.4 Reference parameters. 20
3.5 Returning values from functions. 22

4. The signal routine. 25

5. The Comal structure and call of internal routines in AmigaCOMAL. . . 29

6. Call of procedures and functions written in AmigaCOMAL. 31

7. Programming packages in C. 35

8. Redirection of internal data streams in AmigaCOMAL. 41
8.1 Receiving screen output. 41
8.2 Supplying keyboard input. 42
8.3 Set and return cursor position. 44
8.4 Supplying error texts. 44
8.5 Receiving error output. 46

9. Creating new standard IO devices. 49

10. Signals and exceptions. 53

11. Communication between packages. 55

Appendix A. 57
Variables in AmigaCOMAL 57

Appendix B. 61
Procedure head. 61

Appendix C. 63
Data structures in AmigaCOMAL. 63
C.1 The package structure. 63
C.2 Comal structure. 65
C.3 IO structure. 66
C.4 Device structure. 67
C.5 Exception structure. 68

Appendix D. 69
Calling AmigaCOMAL routines 69
D.1 Calling AmigaCOMAL routines from assembler. 69
D.2 Call of AmigaCOMAL routines from C. 73

Appendix E. 75
Include-files. 75
E.1 The assembler include file Package.i. 75
E.2 The C include file comal.h. 78

Appendix F. 83
References. 83

AmigaCOMAL:The Developers System

Produced by

Svend Daugaard Pedersen

AmigaCOMAL:The Developers System Manual

Written by
Svend Daugaard Pedersen
Len Lindsay

1. Making Stand Alone AmigaCOMAL Programs.

1.1 Introduction.

Running an AmigaCOMAL program is normally very easy. Having completed the development of a program the program is saved on disk using the SAVE command. After that the program can be started just by clicking the mouse on the icon that is automatically created when a program is saved by the SAVE command. If the last statement of the program is BYE you will automatically return to the WorkBench at the end of the program execution.

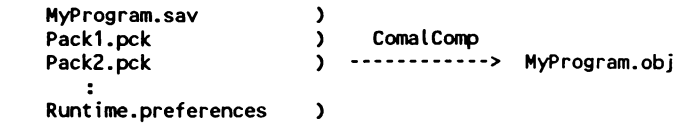
This simple way to start the program works very well for small programs but for larger programs that use a lot of packages it is necessary to have all the packages that the program uses. This makes it a little bit complicated to copy and distribute such programs since you may not know which packages the program uses.

This problem can be solved by using the AmigaCOMAL compiler that makes one stand alone program out of an AmigaCOMAL program and all the packages it uses.

1.2 The AmigaCOMAL compiler ComalComp.

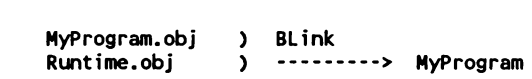
The AmigaCOMAL compiler ComalComp takes as its input an AmigaCOMAL SAVE file (ie, name.SAV). It links this program together with all the packages used by the program and the installation file (if specified) and makes one object code file which is the output of the compiler.

This may be illustrated in the following way:



To make an executable file out of this object code file you have to link it together with the file Runtime.obj. This is done by using the linker BLink (or another linker).

This may be illustrated in the following way:



ComalComp as well as BLink can only be started from the CLI. The compiler is started using the following command:

```
ComalComp [program name] [options]
```

The brackets are omitted.

Example: To compile the program MyProgram.sav, you may use the command:

```
ComalComp MyProgram
```

or just:

```
ComalComp
```

If the last form is used, ComalComp will ask for the name of the program to be compiled.

Note that you should not use the file type .sav. This is set by ComalComp.

It is possible to give supplementary information to ComalComp by using compiler options. Such options consists of a hyphen (-) followed by a letter and

in some cases additional text. Different options are separated by a space. The possible options are:

-p programdir

programdir is the directory (or volume) containing the SAVE file to be compiled. If the -p option is omitted, ComalComp searches for the program in the current directory and if not found there, next in :Programs.

-o outputdir

outputdir is the directory (or volume) where ComalComp places the object code file. If the -o option is omitted the object code will be placed in the current directory.

-u usedir

usedir is the directory (or volume) where ComalComp searches for a package if there is no FROM part in the corresponding USE statement. If the -u option is omitted or if there is no FROM part, the package is searched for in the current directory and if not found there, next in :Packages.

-i [file name]

If the -i option is specified, user defined installation parameters are read. Without the file name, the file Runtime.preferences is searched for in the current directory.

Example: The command:

```
ComalComp MyProgram -o ram: -i
```

compiles the program MyProgram in the current directory (or in :Programs). Packages are searched for in the directory specified in a FROM part if this is present, or otherwise in the current directory or in :Packages. The installation file Runtime.preferences in the current directory will be read. The output file is placed in RAM:.

Example: The command:

```
ComalComp Demo -u df2:Packages -i DEVS:AmigaCOMAL.preferences
```

compiles the program Demo.sav in the current directory (or in :Programs). Packages are searched for in the directory specified in a FROM part if this is present or otherwise in df2:Packages. The installation parameters are read from the file AmigaCOMAL.preferences in the DEVS: directory. The output file is placed in the current directory.

The linker is started by a command such as:

```
BLink Runtime.obj MyProgram.obj to MyProgram
```

For further information see BLink.doc on the distribution disk.

1.3 Combined compilation and linking using the Compile.

The program Compile executes both compiling and linking. The command format is the same as that of ComalComp.

The program will first start ComalComp and then BLink. These two programs must be in the current directory. Compiler options are transferred to ComalComp without changes. The -o option should be avoided since BLink will not be able to find the object code file.

Neither ComalComp nor BLink are able to handle spaces in file names or in directory names.

Example: The command:

```
Compile Demo -p df2:Programs -u df2:Packages
```

works like the two commands:

```
ComalComp Demo -p df2:Programs -u df2:Packages
BLink Runtime.obj Demo.obj to Demo
```

The program Compile makes an icon for the compiled program.

1.4 Compiling from WorkBench.

The program Compile may be started from WorkBench. First click once on the icon for Compile, press <space> and then click twice on the icon for the AmigaCOMAL SAVE file. The two icons need not be in the same drawer.

Compiler options may be placed in the TOOLTYPES array of the Compiler icon. They are written precisely as in the CLI command lines.

2. Package Development.

2.1 Introduction.

AmigaCOMAL is a modern programming language allowing modular programming techniques. In developing larger programs you may divide the program into smaller parts and store these parts as packages.

There are great advantages in such a division. The program is much easier to survey because unimportant programming details are hidden in the packages. Further more, by making the right division, the program will be easier to test since each part may be tested separately.

An additional advantage of using packages to divide the program is that you may later rewrite time critical parts of the program in machine code by using an assembler or a C compiler. This may be done without changing the rest of the program.

Procedures and functions in machine coded packages normally execute faster than procedures and functions written in AmigaCOMAL. However, AmigaCOMAL is very fast by itself so it is not very often you need to rewrite AmigaCOMAL packages into machine code for that reason. The primary reason to write machine coded packages is that in machine code -and particularly in assembler - it is possible to make low level programming in a way which it is not possible in a high level language like AmigaCOMAL.

This part of the development manual describes the development of machine coded AmigaCOMAL packages. To understand the description, it is necessary to have a basic knowledge of the 68000 assembler and the C language. To understand some of the examples it is also necessary to have some knowledge of the Amiga operating system.

2.2 An example.

To get an idea of how a machine coded package is made we will begin with a small example.

We will try to rewrite the following Comal package in machine code.

```
0010 // Comal package
0020
0030 EXPORT even(), odd()
0040
0050 FUNC even(i#) CLOSED
0060   i#:=i# BITAND 1
0070   RETURN i# BITXOR 1
0080 ENDFUNC even
0090
0100 FUNC odd(i#) CLOSED
0110   RETURN i# BITAND 1
0120 ENDFUNC odd
```

The package contains two functions. The function even returns the value true (1) if the argument i# is even and false (0) if i# is uneven. For the function odd it is opposite.

Naturally the machine coded package has to contain the code for the two functions even and odd. They look like:

```
Even:  MOVE.L  -(A5),D3    ; Get argument
       MOVE.L  -(A5),D2    ; .. to D2
       AND.L   #1,D2       ; Return bit 0
       EOR.L   #1,D2       ; .. inverted
       RTS

Odd:   MOVE.L  -(A5),D3    ; Get argument
       MOVE.L  -(A5),D2    ; .. to D2
       AND.L   #1,D2       ; Return bit 0
       RTS
```

In addition to this code, it is necessary to tell AmigaCOMAL where the code is, what their names are, which parameters they have etc. This information is collected in a so called "procedure head" (we use the word "procedure" even in this case where it is in fact a function). The procedure itself we will call the "procedure body". The procedure head looks like:

```
EvenLink:    DC.L   0            ; Last procedure
             DC.L   EvenName     ; Address of name
             DC.W   bytecode     ; Function of type byte
             DC.W   1            ; One argument
             DC.W   longval      ; Argument type = long value
             DC.L   Even         ; Routine address
EvenName:    DC.B   'EVEN',0

OddLink:     DC.L   EvenLink
             DC.L   OddName
             DC.W   bytecode     ; Function of type byte
             DC.W   1            ; One argument
             DC.W   longval      ; Argument type = long value
             DC.L   Odd          ; Routine address
OddName:     DC.B   'ODD',0

LinkTop:     DC.L    OddLink
```

Note, that the two procedure heads are linked together by the first long word in each procedure head. The start of the chain of procedure heads is found in the address LinkTop. This address has to be transferred to AmigaCOMAL. This is done in the initialization routine.

This initialization routine which is called immediately after the package is read from disk, must be in the start of the package. In the simple package we are going to make it looks like:

```
; Initialization routine
   MOVE.L  D0,A0                ; Address of package structure
   MOVE.L# LinkTop,PckTopId(A0) ; Set address of chain
   CLR.W   D0                   ; Everything ok!
   RTS
```

Immediately before the call to the initialization routine AmigaCOMAL places the address of a special package structure in D0.L. This structure contains a number of information about the package. One is the address of the first link in the chain of procedure heads. It is the task of the initialization routine to place this address in the right place in the package structure. Before the return from the initialization routine the status code must be placed in D0.W. In this case there is no error and we return a zero in D0.W.

The complete package looks like:

```
; EvenOdd - version 20.12.88

   include 'ram:package.i'          ; Standard definitions

; Initialization section
   MOVE.L  D0,A0                    ; Address ofpackage str
   MOVE.L  #LinkTop,PckTopId(A0)    ; Set top link address
   CLR.W   D0                       ; No error
   RTS

; FUNCtion even
;
Even:          MOVE.L  -(A5),D3             ; Get arguments
               MOVE.L  -(A5),D2
               AND.L   #1,D2
               EOR.L   #1,D2
               RTS

; FUNCtion odd
;
Odd:           MOVE.L  -(A5),D3             ;Getarguments
               MOVE.L  -(A5),D2
               AND.L   #1,D2                ;Return bit zero
               RTS

      SECTION EvenOddData,DATA

EvenLink:      DC.L    0                    ; Last name
               DC.L    EvenName
               DC.W    bytecode             ; Byte integer fct
               DC.W    1 ; One argument
               DC.W    longval              ; Argument type = long
               DC.L    Even                 ; Routine address
EvenName:      DC.B    'EVEN',0

OddLink:       DC.L    EvenLink
               DC.L    OddName
               DC.W    bytecode             ; Byte integer fct
               DC.W    1                    ; One argument
               DC.W    longval              ; Argument type = long
               DC.L    Odd                  ; Routine address OddName:
               DC.B    'ODD',0

LinkTop:       DC.L    OddLink
```

In the example we have used some symbols (bytecode, longval etc.). They are, along with other symbols, defined in an include file Package.i which is found on t he distribution disk.

The complete package may be assembled and linked by using the execute file genpack. Then it may be used like any other package. Seen from a users point of view there is no difference between a machine coded package and a package written in AmigaCOMAL.

3. Internal format of data in AmigaCOMAL.

In the last section we did not discuss how numbers are represented in AmigaCOMAL, how parameters are transferred by a call to a package routine, how values are returned, etc. We will do this in the following sections.

3.1 Number format in AmigaCOMAL.

In AmigaCOMAL a number always occupies two 32 bits registers. The primary register pair is D2,D3 which we will call NUMREG2, and the secondary is D0,D1 which is called NUMREG1. The number format is:

```
D3.W <> 0 :  the number is represented as floating point
D2.L is a 32 bit mantissa
D3.W is the exponent + $8000

D3.W = 0 :   the number is an integer (long, short or byte) contained in
             D2.L (possibly sign extended)
```

For a number in floating point format the most significant bit of the mantissa (which is always 1) is used as a sign bit.

Example: The number 3 may be represented as:

```
   D2.L=$40000000 ,  D3.W=$8002 (floating point format)

or

   D2.L=$00000003 ,  D3.W=$0000 (integer format)

and the number -3.14159265 as:

   D2.L=$C90FDAA2 ,  D3.W=$8002
```

Naturally the last number can only be represented as floating point.

It is seen that a 48 bits floating point format is used. The high part of register D3 is unused. It is reserved for a later shift to IEEE 64 bit floating point format.

3.2 Registers and use of storage.

With every call to a package routine the register A6 points to the top and register A5 points to the bottom of the free RAM. All RAM between these two addresses may be used freely as temporary storage. In addition, the address register A5 serves as a pointer to the parameter stack (see next section).

All registers may be used by a package routine, but if the routine returns by using the return address on top of the CPU stack (for instance by using a RTS instruction) the content of the stack pointer (register A7) must be reestablished.

Example: A code like the following may not be used:

```
:
MOVE.L  (A7)+,RetAddr      ; Save return address
:
:      (one or another code changing A7)
:
MOVE.L  RetAddr,A0         ; Get return address
JMP     (A0)               ; .. and return to COMAL
```

If it is necessary to return to AmigaCOMAL from a place where the content of register A7 is unknown, you should use the special return routine (see appendix D and the example in section 3.4).

3.3 Transferring of parameters.

Before a call to a function or a procedure, AmigaCOMAL moves the parameters onto the parameter stack (A5). The order is that of the formal parameters in the procedure head. Consequently the last parameter is on top of the stack and is the first one to take down again.

The format of the parameter depends on the type.

Value parameters.

When a number appears as a value parameter, the two long words that a number always occupy, are placed on the stack. The number may be taken down from the stack by a code fraction like:

```
MOVE.L   -(A5),D3          ; Get exponent
MOVE.L   -(A5),D2          ; .. and mantissa
```

We have already seen an example of this in section 2.2.

For strings it is a little more difficult. AmigaCOMAL calculates the value of the string expression which is the actual parameter and the result is placed in the work space (the RAM between A5 and A6). The address of the start of the text and the length of the text are then pushed onto the stack.

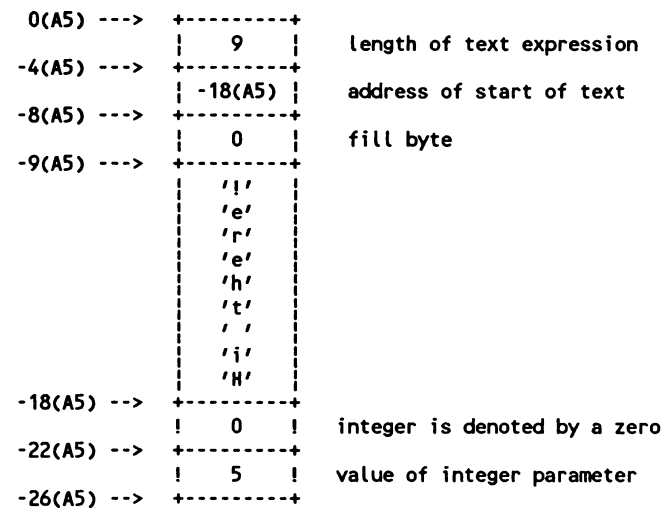
Example: Let us say we have a procedure that in AmigaCOMAL would be defined as:

```
PROC alfa(n#,text$)
```

If this procedure is called from the line:

```
alfa(2+3,"Hi"+" there"+"!")
```

the stack will look like:



Thus you can get the parameters from the stack in the following way:

```
MOVE.L   -(A5),D1              ; Get length of text to D1.W
MOVE.L   -(A5),A2              ; .. and address to A2
MOVE.L     A2,A5               ; Move stack beyond text value
MOVE.L   -(A5),D3
MOVE.L   -(A5),D2              ; Get the integer value
```

Note that although the length of the text is contained in a long word, it is only the least significant 16 bits that contain the length.

Records and arrays cannot appear as value parameters in a package procedure. Use reference parameters instead.

3.4 Reference parameters.

Reference parameters always occupy three long words on the parameter stack. If these long words are taken down from the stack with the following code fraction:

```
MOVE.L   -(A5),D3              ; Get type
MOVE.L   -(A5),A3              ; .. address of structure information
MOVE.L   -(A5),A2              ; .. and address of data
```

the register D3 will contain (as seen from the comment) the type of the variable, register A3 will contain the address of the structure information of this variable and A2 will contain the address of the content of this variable (the data field). The structure information address is only used if the variable is an array or a record (but the word is always present on the stack!).

The real type is found in D3.B. The bits 8-15 in D3 contain the so called type attributes. The most significant 16 bits in D3 are not used.

The possible type values are listed in appendix A along with a precise description of the structure information and the data field of a variable.

Example: Let us see how this function is written in assembler:

```
FUNC strmax(REF t$)
```

The function should return the maximal length of a string variable.

```
; Maximal string length package - version 23.12.88

include 'ram:package.i'
```

```
   ; Initialization section
            MOVE.L  D0,A0                   ; Package structure
            MOVE.L  #LinkTop,PckTopId(A0)   ; Address of top link
            MOVE.L  PckCmlStr(A0),ComalStr  ; Comal structure
            CLR.W   D0                      ; No error
            RTS

   ; String max function
MaxLen:     MOVE.W  -2(A5),D1           ; Get type
            CMP.B   #strg,D1            ; String ?
            BNE.S   ARGERR              ; No -> error
            MOVE.L  -12(A5),A2          ; Get address of string var
            BTST    #POINTERBIT,D1      ; Pointer ?
            BEQ.S   MaxLen1             ; No -> branch
            MOVE.L  (A2),A2             ; Get data address
MaxLen1:    MOVE.W  (A2),D2             ; Get maximal string length
            EXT.L   D2                  ; Extend to long
            CLR.L   D3                  ; Result is an integer
            RTS                         ; Return

   ; Error
ARGERR:     MOVEQ   #76,D0          ; Type error
            MOVE.L  ComalStr,A0     ; Get Comal structure
            JMP     CMLRET(A0)      ; Return error

            SECTION MaxLenData,DATA

MaxLenLink: DC.L    0
            DC.L    MaxLenName
            DC.W    shortcode       ; Short integer function
            DC.W    1               ; One argument
            DC.W    strgref         ; REF - text
            DC.L    MaxLen          ; Routine address
MaxLenName: DC.B    'MAXLEN',0

LinkTop:    DC.L    MaxLenLink

            SECTION MaxLenBlock,BSS
   ;
   ; Internal workspace
   ;
ComalStr:   DS.L    1               ; Pointer to Comal structure

            END
```

The function first gets the type from the stack and tests if it is a string. Then the address of the data field is taken down and it is examined to see if it is a pointer variable. If this is the case, the data field contains the address of the content and not the content itself. The maximal length is read into D2 and this value is returned in NUMREG2.

If the type is not correct an error code is placed in D0 and the function returns to AmigaCOMAL through a special jump address. More about that in section 5.

3.5 Returning values from functions.

Functions return values in the register pair D2,D3. In sections 2.2 and 3.1. we have seen examples of returning numbers. In this case the register pair is simply treated as a number register (NUMREG2) with the format described in section 3.1.

If it is a string function the length of the returned text is placed in D3.W and the address in D2.L. The text itself may be anywhere in RAM but normally it is placed in the work space (pointed at by A5).

Example: The function bin$ with the AmigaCOMAL format

```
FUNC bin$(n#)
```

should return a string representing the value of the argument n# written as a binary number (preceded by a %). A package with this single function is made in this way:

```
; Bin package - version 23.12.88

      include 'ram:package.i'

; Initialization section
      MOVE.L  D0,A0       ; Package structure
      MOVE.L  #LinkTop,PckTopId(A0)    ; Address of top link
      CLR.W   D0          ; No error
      RTS

; Return number as binary string
Bin:  MOVE.L  A5,D2       ; Address of string to D2
      MOVE.B  #'%',(A5)+
      MOVEQ   #30,D4      ; Digit counter
      MOVE.L  -9(A5),D1   ; Get number
      BRA.S   Bin2        ; .. and branch into loop
Bin1: ADD.L   D1,D1
Bin2: DBMI    D4,Bin1     ; Loop until 1. non zero bit
      ADDQ.W  #1,D4
      CLR.B   D5
Bin3: MOVE.B  #'0',D0
      ADD.L   D1,D1       ; Digit value into carry
```

```
          ADDX.B  D5,D0
          MOVE.B  D0,(A5)+    ; Set character of string
          DBRA    D4,Bin3
BinEnd:   MOVE.L  A5,D3
          SUB.L   D2,D3       ; Calculate length
          RTS                 ; .. and return

          SECTION BinData,DATA

BinLink:  DC.L    0
          DC.L    BinName
          DC.W    strgcode    ; String function
          DC.W    1           ; One argument
          DC.W    longval     ; Arg type = long int value
          DC.L    Bin         ; Routine address
BinName:  DC.B    'BIN$',0

LinkTop:  DC.L    BinLink

          END
```

The string is built up in the work space, the start of which is found in A5. For that reason the address of the final string is moved to D2 at the start of the function. Having made the whole string, the length is calculated and this length is placed in D3.

4. The signal routine.

Like Comal packages, machine coded packages may receive signals. To tell AmigaCOMAL that a package wants signals sent, it has to put the address of the signal routine into the package structure. This is normally done in the initialization routine.

An initialization routine that sets this address may look like:

```
; Initialize routine
   MOVE.L  D0,A0          ; Address of  pck structure
   MOVE.L  #LinkTop,PckTopId(A0)    ; Set address of chain
   MOVE.L  #Signal,PckSignal(A0)    ; .. and of signal routine
   CLR.W   D0             ; Every thing ok!
   RTS
```

Before a call to a signal routine AmigaCOMAL places the signal number in D0.W. The signal routine may like any other package routine use all registers and it may perform all necessary actions like closing libraries, release allocated RAM, change the package structure etc.

Example: The following package contains one single function:

```
   FUNC translate$(t$)
```

By using the translator.library, it translates an English text into a special phonetic script.

```
; Translator package - version 24.12.88

      include 'ram:package.i'
      include ':Include/Exec.i'

; Equates
ExecBase:       EQU  4
_LVOTranslate:  EQU  -30

; Initialization section
      MOVE.L   D0,A0                  ;Package structure
      MOVE.L   #LinkTop,PckTopId(A0)  ; Address of top link
      MOVE.L   #Signal,PckSignal(A0)  ; Address of Signal
      MOVE.L   ExecBase,A6
      CLR.L    D0
      LEA      TransLibName,A1
      JSR      _LVOOpenLibrary(A6)    ; Open translator lib
      MOVE.L   D0,TransBase           ; Store base address
```

```
BNE.S    InitOk                  ; Ok if not zero
MOVE.W   #199,D0                 ; Initialization err
BRA.S    InitRet
InitOk:   CLR.W    D0            ; No error
InitRet:  RTS

; Signal routine
Signal:   CMP.W    #2,D0                 ; DISCARD ?
      BEQ.S    CloseTrans                ; Yes -> close translator lib
      CMP.W    #10,D0                    ; BYE ?
      BEQ.S    CloseTrans                ; Yes->close translator lib
SignalRet:   RTS


CloseTrans:  MOVE.L  ExecBase,A6
      MOVE.L  TransBase,D0               ; Get lib base
      BEQ.S   SignalRet
      MOVE.L  D0,A1
      JSR     _LVOCloseLibrary(A6)       ; Close library
      CLR.L   TransBase
      RTS

; Translator function
Translate:   MOVE.L  -(A5),D3            ; Get length ..
      MOVE.L  -(A5),D2                   ; .. and address of string
      TST.W   D3                         ; Zero length ?
      BEQ.S    TranslateRet              ; Yes -> return
      MOVE.L  D2,A0                      ; Address of string to A0
      MOVE.L  D3,D0                      ; .. and length to D0
      MOVE.L  A5,A1                      ; Buffer address
      MOVE.L  #512,D1                    ; .. and length
      MOVE.L  TransBase,A6
      JSR     _LVOTranslate(A6)          ; Translate
      MOVE.L  A5,D2                      ; Addr. of translated string
      MOVE.L  A5,D3                      ; Start calculating length 1$:
      TST.B   (A5)+
      BNE.S   1$                         ; Loop until end
      SUB.L   A5,D3
      NOT.L   D3                         ; D3.L = length
TranslateRet: RTS                        ; Return

      SECTION TranslateData,DATA

TranslateLink: DC.L    0
      DC.L   TransName
      DC.W   strgcode                    ; String function
      DC.W   1                           ; One argument
      DC.W   strgval                     ; Text value
      DC.L   Translate                   ; Routine address
TransName:   DC.B   'TRANSLATE$',0

LinkTop:  DC.L  TranslateLink

TransLibName:    DC.B    'translator.library',0
```

```
        SECTION TranslateBlock,BSS
;
; Internal workspace
;
TransBase:   DS.L  1                ; Translator base

             END
```

In the initialization routine, the translator.library is opened. If it fails, the error code 199 is returned. The signal routine tests for BYE and DISCARD signals. If these signals are received the library is closed.

The translated text may be sent to the narrator.device to be pronounced.

It should be remarked that the Translator_library on the AmigaCOMAL system disk is different from this package.

5. The Comal structure and call of internal routines in AmigaCOMAL.

The address of the comal structure which is a data area where AmigaCOMAL stores important system information (see appendix C.2), is found in the package structure. If you need this address you will normally get it and store it during the initialization (see the example in section 3.4).

Just below the comal structure a jump table is found. This table makes it possible to call useful routines inside AmigaCOMAL. We have already used this jump table ta call a return routine (with status code in D0.W). In appendix D you will find a complete list of all the routines to be called.

Example: The following package shows how the mathematical routines in AmigaCOMAL are used. The single function in the package calculates cotangents to an angle

```
; Math package - version 24.12.88

   include 'ram:package.i'

; Initialization section
   MOVE.L   D0,A0                      ; Address of package structure
   MOVE.L   #LinkTop,PckTopId(A0)      ; Address of top link
   MOVE.L   PckCmlStr(A0),ComalStr     ; Comal structure
   CLR.     WD0                        ; No error
   RTS

; Return value of cot(x)=cos(x)/sin(x)
;
Cot:   MOVE.L   -(A5),D3               ; Get exponent
       MOVE.L   -(A5),D2               ; .. and mantissa of x
       MOVE.L   D3,D1                  ; Move to NUMREG1
       MOVE.L   D2,D0
       MOVE.L   ComalStr,A0            ; Address of Comal structure
       JSR      NUMSIN(A0)             ; NUMREG2:=sin(NUMREG2)
       EXG      D0,D2                  ; NUMREG1 <-> NUMREG2
       EXG      D1,D3
       JSR      NUMCOS(A0)             ; NUMREG2:=cos(NUMREG2)
       JMP      NUMDIV(A0)             ; NUMREG2:=NUMREG2/NUMREG1

       SECTION MathData,DATA

CotLink: DC.L   0
         DC.L   CotName
```

```
        DC.W   fltcode                  ; Float function
        DC.W   1                        ; One argument
        DC.W   fltval                   ; Argument type = float value
        DC.L   Cot                      ; Routine address
CotName: DC.B  'COT',0

LinkTop: DC.L  CotLink

        SECTION MathBlock,BSS
;
; Internal workspace
;
ComalStr: DS.L  1                       ; Pointer to Comal structure
```

The function does not test if cotangents is defined for the actual parameter. If sin(x) is zero NUMDIV will automatically report an error (Division by zero). If you want another message you have to test the value of sin(x). This can be done by calling the COMAL routine TSTSGN.

Sometimes a package routine has tasks that must be terminated before it is left. If an AmigaCOMAL routine does not return because of an error, you have to use the signal routine clean up.

6. Call of procedures and functions written in AmigaCOMAL.

Among the routines to be called through the jump table there is one by name CallUser. This routine allow you to call closed functions and procedures written in AmigaCOMAL.

Before the call to this routine the actual parameters of the procedure/function to be called are placed on the parameter stack, the address of the field in the procedure head containing the number of parameters is placed in D2.L and the type in D3. The registers A5 and A6 must point to the start and the end of a free work space (at least 1Kb).

The address and the type of the function/procedure to be called may be transferred as value parameters.

Example: The following little program calls a package routine test:

```
0010 FUNC inverse(x) CLOSED
0020   RETURN 1/x
0030 ENDFUNC inverse
0040
0050 PRINT test(inverse())
```

Let us imagine that the function test is defined by the following procedure head:

```
TestLink:    DC.L  0
             DC.L  TestName
             DC.W  fltcode                ; Machine coded float2 function
             DC.W  1                      ; One argument
             DC.W  fltref                 ; REF - float (function)
             DC.L  Test                   ; Address of routine
TestName:    DC.B  'TEST',0
```

The routine test may get the necessary information about the actual parameter in this way:

```
Test: MOVE.L  -4(A5),D3              ; Get type
      MOVE.L  -12(A5),A2             ; .. and address
```

Now the register A2 points to the number of parameters in the procedure head for the function inverse. The type of each parameter follows this number just like it is the case for a procedure written in machine code.

In the procedure head for the function test the parameter type is specified as fltref. This means that number variables, number arrays, number functions and procedures are accepted by AmigaCOMAL. Having got the address and the type, the function has to test for correct type. In this example it would be natural to accept functions written in AmigaCOMAL as well as machine coded functions (like the function sin or a function in a machine coded package). If it is a function written in AmigaCOMAL you have to ensure that it is closed.

All these tests may be done in this way:

```
CMP.B  #fltcode,D3                ; Machine coded function ?
BEQ.S  Test1                      ; Yes -> branch
CMP.B  #fltuser,D3                ; Comal function ?
BNE.S  ARGERR                     ; No -> return error
AND.W  #%0011100000000000,D3      ; Clsd/external/package ?
BEQ.S  ARGERR                     ; Report error if not
```

Next we have to test for correct number and types of the parameters. In this case it must be a function with one value parameter (floating point) so the test may be done in a single instruction followed by a conditional jump:

```
Test1:   CMP.L  #$0001002F,(A2)   ; Correct number/type of arg ?
         BNE.S  ARGERR            ; No -> return error
```

Now we are ready to call the function. A call to this with the number 3 as the argument may be done in this way:

```
MOVE.L  -4(A5),D3                 ; Get type
MOVE.L  -12(A5),D2                ; .. and address
MOVE.L  #3,(A5)+                  ; Number onto ..
CLR.L   (A5)+                     ; .. the parameter stack
MOVE.L  ComalStr,A0                 ; Get address of comal struct
JSR     CallUser(A0)              ; Call the function
SUBQ.L  #8,A5                     ; Remove parameter from stack
:                                 ; The value of the Function
:                                 ; .. is now in NUMREG2
```

CallUser preserves the registers A5 and A6. If the call is to a function the value will be returned in the register pair D2,D3 as specified in section 3.5. The contents of all other registers are unknown.

A routine which is calling procedures or functions has to satisfy some extra demands.

First of all it has to be re-entrant which means that you may not use fixed addresses to store temporary values. Use either the CPU stack or the parameter stack (normally the last is the most roomy).

Second it may cause problems if an error occurs in the routine which is called. In this case there is no return. The problem is the same as was mentioned in the previous section.

7. Programming packages in C.

Whether it is developed in assembler or in C, a machine coded package must have the format described in the previous sections.

To make it easier to develop packages in C, a new startup module p.o (to be used instead of c.o during the linking), a special library module comal.lib (to be used along with other library modules) and an include file comal.h is supplied. All these modules are developed for the Lattice C v. 3.03 compiler and is found on the distribution disk.

The following package shows how the modules p.o, comal.lib and comal.h are used. The package contains two functions cot and inv.

```
/*****************************************************************************/
/*                                                                           */
/*                    Math package  -  version 30.12.88                      */
/*                                                                           */
/*****************************************************************************/

#include "comal/comal.h"

extern CmlFlt CmlCos(), CmlSin();

int c1 = 1;

void cot(wrkspc,wrktop)                 /* cot(x) = cos(x)/sin(x)            */
CmlFlt  *wrkspc,*wrktop;                /* pointers to workspace             */
{
   CmlFlt x,value;                      /* Floating point variables          */

   WorkBot = (APTR)wrkspc;              /* Set work space bottom ..          */
   WorkTop = (APTR)wrktop;              /* .. and top                        */

   x = *(--wrkspc);                     /* x = actual parameter              */
   value = CmlCos(x)/CmlSin(x);
   RetValue(&value);                    /* Return                            */
}

void inv(wrkspc,wrktop)                 /* inv(x) = 1/x                      */
CmlFlt  *wrkspc,*wrktop;                /* pointers to workspace             */
{
   CmlFlt x,value;                      /* Floating point variables          */

   WorkBot = (APTR)wrkspc;              /* Set work space bottom ..          */
   WorkTop = (APTR)wrktop;              /* .. and top                        */
```

```
    x = *(--wrkspc);                    /* x = actual parameter              */
    value = c1/x;
    RetValue(&value);                   /* Return                            */
}


                                        /***********************************/
struct varnode invlnk = {               /* Identifier table for inv          */
    0,                                  /* Last link                         */
    "INV",                              /* Name                              */
    fltcode,                            /* Type                              */
    1       };                          /* one argument                      */
UWORD invpar1 = fltval;                 /* Float value parameter             */
void (*invrout)() = &inv;               /* Routine address                   */
                                        /***********************************/


                                        /***********************************/
struct varnode cotlnk = {               /* Identifier table for              */
                                        /* cotangents                        */
    &invlnk,                            /* Link to next procedure head       */
    "COT",                              /* Name                              */
    fltcode,                            /* Type                              */
    1       };                          /* one argument                      */
UWORD cotpar1 = fltval;                 /* Float value parameter             */
void (*cotrout)() = &cot;               /* Routine address                   */
                                        /***********************************/

topptr topnode = &cotlnk;               /* Top link points to cotlnk         */

void init(pck)
struct package *pck;
{
pak->var = &topnode;                    /* Initialize top-id link            */
}
```

In the start of the program the file comal.h is included. The two functions from comal.lib used in this program are declared.

The functions (and procedures) that are to be exported from a C package are always made as procedures with two parameters: the top and bottom of the work space. Before a package routine is called AmigaCOMAL places the registers A5 and A6 on the CPU stack. Thus the content of these registers may be read as the parameters of the C procedure.

The first thing done by the procedures of our package is to store the value of the two parameters in the global variables WorkTop and WorkBot. These variables are declared in the file comal.h. It is necessary to do this if the procedures are calling the AmigaCOMAL routines (in this case CmlCos() and CmlSin()), since the interface routines in comal.lib read the address of the work space in these variables. The only AmigaCOMAL routines not using these

addresses are RetValue, RetStat and the two routines used to call procedures and functions: CallProc and CallFunc.

Values from functions are returned by calling RetValue defined in comal.lib. RetValue must have a pointer as parameter (in this case CmlFlt). The pointer must point to a data area consisting of two long words (corresponding to the registers D2.L and D3.L).

Numbers from AmigaCOMAL are declared as CmlFlt. Interface routines in comal.lib makes it possible to use the floating point format used by AmigaCOMAL. But there are some limitations. First of all only the algebraic operators (+,-,*,/ and monadic -) and the relational operators (<,<=,>,>=,== and !=) may be used. Next, it is not possible to use constants in a floating point expression. However, integer variables are valid.

The initialization routine must have the name init. It has a pointer to the package structure as parameter. Naturally init should do the same initialization as an initialization routine in an assembler programmed package, i.e. as a minimum init should place the address of the chain of procedure heads in the package structure.

The signal routine (not found in the example) is a procedure with the signal number as parameter. A typical signal routine looks like:

```
void signal(s)
short s;
  {
   switch(s)
   {
     case 1:
      :
      :
   case 10:
   };
  };
```

The address of the signal is set in the initialization routine in the following way:

```
pck->signal = &signal;
```

This address may not be changed after the initialization!

In the programming manual we made a mathematical package which

contained a special integral function. We mentioned that it would be possible to write this function in machine code. We are now able to write this function in C. It may look like:

```
/*****************************************************************************/
/*                                                                           */
/*        Integral package  -  version 27.01.89                              */
/*                                                                           */
/*****************************************************************************/

#include "comal/comal.h"

extern CmlFlt NumFunc();

int c2 = 2, c3 = 3, c6 = 6;

#define f(x)  (NumFunc(fnct,x))

void integral(wrkspc,wrktop)          /* integral(fnct(),a,b)                */
CmlFlt  *wrkspc,*wrktop;              /* pointers to workspace               */
{
   struct refparam *fnct;
   CmlFlt a,b,xi,dx,sum;              /* Floating point variables            */
   int i,n;
   ULONG *arg;                        /* Pointer to arguments                */

   b = *(--wrkspc); a = *(--wrkspc);
   fnct = (struct refparam *)wrkspc;
   (--fnct);
   if (fnct->Type!=fltuser && fnct->Type!=fltcode) RetStat(76);
   if (fnct->Type==fltuser)
     if (fnct->Attributes & (IMPORTBIT | CLOSEDBIT | EXTERNALBIT)==0)
       RetStat(76);
   arg = (ULONG *)fnct->Data;
   if (*arg != 0x01002F) RetStat(76);
   WorkBot = (APTR)wrkspc;            /* Set work space bottom ..            */
   WorkTop = (APTR)wrktop;            /* .. and top                          */

   n = 32;
   dx = (b-a)/n;
   xi = a; sum = (f(a)-f(b))*dx/c6;
   for (i = 1; i<=n; i++)
     {
       xi = xi+dx;
       sum = sum+(f(xi)+c2*f(xi-dx/c2))*dx/c3;
     };
   RetValue(&sum);
}
```

```
                                    /**********************************/
struct varnode intlnk = {           /* Identifier table for integral  */
0L,                                 /* Last link                      */
"INTEGRAL",                         /* Name                           */
fltcode,                            /* Type                           */
3};                                 /* one argument                   */
UWORD intpar1 = fltref;             /* Float reference parameter      */
UWORD intpar2 = fltval;             /* Float value parameter          */
UWORD intpar3 = fltval;             /* Float value parameter          */
void (*introut)() = &integral;      /* Routine address                */
                                    /**********************************/

topptr topnode = &intlnk;           /* Top link points to cotlnk      */

void init(pck)
struct package *pck;
{
pck->var = &topnode;                /* Initialize top-id link         */
}
```

Having compiled and linked this package we have to remove the integral function from the original mathematical package and in the start of this package add the line

```
USE INTEGRAL
```

Now the mathematical package may be used as before. It is just a little bit faster.

If the speed is still not satisfactory you may try to rewrite it in assembler. But this is much more difficult.

8. Redirection of internal data streams in AmigaCOMAL.

Until now we have seen how ordinary procedures and functions are written in machine code. But from a machine coded package it is possible to intervene in the AmigaCOMAL system itself.

The package structure contains some flags (see appendix C.1). By setting one or more of these flags the package will receive or be requested to deliver certain data streams. These data streams are screen output, keyboard input, cursor position, delivering of error texts and receiving of error output.

8.1 Receiving screen output.

If a package wants to get all output to the screen, it may set the PckConOutBit (bit 0) in the flag field of the package structure and place the address of a ConOut routine in the PckConOut field (offset 24) of the package structure.

Before a text is going to be written on the screen AmigaCOMAL checks to see if one or more packages has requested the receiving of the output. If this is the case the output is send to that (these) package(s) in the order determined by the priority field of the package structure.

At the call to the ConOut routine of a package the address of the text is found in D2.L and the length in D3.L (see also appendix C.1).

The package may do anything with this text but at the return the registers D2.L and D3.L must contain the address and the length of a text (the same as it received or quite another one). If D3.L is non zero this text is send to the next package or to the AmigaCOMAL screen if there is no more packages.

Example : A package that are receiving screen output may look like:

```
   include 'ram:package.i'

; Initialization section
   MOVE.L  D0,A0                       ; Package structure
   MOVE.L  A0,PckStr                   ; Save package
   MOVE.L  PckCmlStr(A0),ComalStr      ; .. and  Comal str
   MOVE.L  #LinkTop,PckTopId(A0)       ; Set top link
   MOVE.L  #ConOut,PckConOut(A0)       ; Address of ConOut
   MOVE.B  #4,PckPrior(A0)             ; Set priority
   BSET    #PckConOutBit,PckFlag+1(A0) ; .. and flag
   CLR.W   D0                          ; Status
   RTS                                 ; Return

; ConOut routine
ConOut:
            :
            :

   RTS

   SECTION ConOutData,DATA

LinkTop: DC.L   0                      ; No procedures

              :
```

In this case the package does not export any procedures so that LinkTop is set to zero. The priority of the package is set to 4. In this case the choice is random but illustrates that in a package of this type the priority is very often set.

The priority as well as the flags may be changed later on. This may be done in the signal routine (see also next section).

8.2 Supplying keyboard input.

There are two types of keyboard input (corresponding to the two AmigaCOMAL standard functions key$ and inkey$). In one of these routines the keyboard is scanned and in the other, the routine will wait until a key is pressed.

If a package wants to deliver keyboard input (one or both sorts) it has to set the relevant flags and the corresponding addresses in the package structure (see appendix C.1).

Example: The following package serves as a keyboard filter, filtering off the Help key. This is done by setting the PckConInBit (bit 1) in the flag field of the package structure.

The ConIn routine in the package calls the AmigaCOMAL ConIn by using the jump table below the comal structure. Having received a key value from AmigaCOMAL it tests to see if it is the Help key. If this is not the case it just returns the received value. Otherwise it writes a help text on the screen and ConIn is called again.

```
; HELP package - version 29.12.88

   include 'ram:package.i'

; Initialization section
   MOVE.L  D0,A0                       ; Package structure
   MOVE.L  A0,PckStr                   ; Save package
   MOVE.L  PckCmlStr(A0),ComalStr      ; .. and  Comal str
   MOVE.L  #LinkTop,PckTopId(A0)       ; Set top link
   MOVE.L  #Signal,PckSignal(A0)       ; .. and Signal rout.
   MOVE.L  #ConIn,PckConIn(A0)         ; Address of ConIn
   MOVE.B  #4,PckPrior(A0)             ; Set priority
   BSET    #PckConInBit,PckFlag+1(A0)  ; .. and flag
   CLR.W   D0                          ; Status
   RTS                                 ; Return

; Signal routine
Signal:       CMP.W   #5,D0             ; Change to command mode ?
              BCC.S   ChgConInFlg       ; Yes -> branch
              RTS

ChgConInFlg:  MOVE.L  PckStr,A0
              CMP.W   #7,D0             ; .. or to execute mode ?
              BCC.S   SetConInFlg       ; Yes -> branch
              BCLR    #PckConInBit,PckFlag+1(A0)  ; Clear ConIn flag
              RTS

SetConInFlg:  BSET    #PckConInBit,PckFlag+1(A0)  ; Set ConIn flag
              RTS


; ConIn routine
ConIn0:       MOVE.L  #HelpText,D2
              MOVE.L  #HelpLen,D3
              JSR     CONOUT(A0)        ; Output Help text

; Routine entrance
ConIn:        MOVE.   LomalStr,A0
              JSR     CONIN(A0)         ; Get character to D0.B
              CMP.B   #$94,D0           ; 'Help' ?
              BEQ.S   ConIn0            ; Yes -> output help text
              RTS
```

```
          SECTION HelpData,DATA

LinkTop:      DC.L    0                       ; No procedures

HelpText:     DC.B      'The Help key is not of much use..'
              DC.B    10,13
              DC.B      '.. but it might be that!',10,13
HelpLen:      EQU       *-HelpText

          SECTION HelpBlock,BSS
;
; Internal workspace
;
ComalStr:     DS.L    1                 ; Address of Comal structure
PckStr:       DS.L    1                 .. and package structure

              END
```

8.3 Set and return cursor position.

A package may set and/or return the position of the cursor by setting flags and addresses in the package structure (see appendix C.1 for further details). Normally this is done only if the package are receiving screen output.

8.4 Supplying error texts.

If a package wants to supply error texts it must set the PckErrTxtBit (bit 7) and the corresponding address in the package structure (offset 48). When AmigaCOMAL is needs to find a text for a given error code it calls the package(s) (if any) that have set the flag PckErrTxtBit. At the call the error code is in D0.L and at the return the address and length of the error text must be in the registers D2.L and D3.L. If D3.L is zero, the next package is called, on until there are no more packages. Then the usual AmigaCOMAL error text is used.

Example: The following C programmed package will deliver the first nine error texts in Danish.

```
/*****************************************************************************/
/*                                                                           */
/*             Dansk - version 29.12.88                                      */
/*                                                                           */
/*****************************************************************************/
#include "comal/comal.h"

#define AND &&
#define OR ||
#define begin {
#define end }

STRPTR ErrTxtTab[] =
  begin
    "001: Ulovligt tegn",
    "002: Ulovligt linjenummer",
    "003: Tekstkonstant for lang",
    "004: Variabel forventet",
    "005: Konstant forventet",
    "006: Taludtryk forventet",
    "007: Tekstudtryk forventet",
    "008: \'(\' forventet",
    "009: \')\' forventet"
  end;

#define maxerr 9

int strlen(s)
STRPTR s;
begin
    int i;
    for (i=0; *s != '\0'; s++) i++ ;
    return i;
end

void ErrTxt(wrkspc,wrktop)
int *wrkspc,*wrktop;
begin
    int err;
    struct valparam ErrorText;

    err = *(--wrkspc);
    if ( err<=maxerr )
      begin
        ErrorText.d2 = (long)ErrTxtTab[err-1];
        ErrorText.d3 = strlen(ErrTxtTab[err-1]);
      end
    else
      ErrorText.d3 = 0;
    RetValue(ErrorText);
end

void init(pck)
struct package *pck;
  begin
```

```
  pck->var = NULL;
  pck->ErrTxt = &ErrTxt;
  pck->Flags  |= ErrTxtFlag;
  pck->Priority = -20;
end
```

8.5 Receiving error output.

You may do more than just deliver error texts. By setting the PckErrOutBit (bit 6) and the corresponding address in the package structure (offset 44) you may decide what is to be done with the text.

Example: The following package sends the error text to the Amiga narrator device.

```
/*****************************************************************************/
/*                                                                           */
/*          SayError package  -  version 31.12.88                            */
/*                                                                           */
/*****************************************************************************/

#include "comal/comal.h"
#include "exec/ports.h"
#include "devices/narrator.h"
                                        /* Exec library                      */
extern APTR OpenLibrary();
extern LONG OpenDevice();
extern void CloseLibrary(),CloseDevice(),BeginIO(),WaitIO();

extern struct MsgPort *CreatePort();    /* Exec support library              */
extern void DeletePort();

struct package *SayError;
struct MsgPort *ReplyPort;
struct narrator_rb narrat;
UBYTE amaps[] = {3,5,10,12};

int busy;
LONG nar_flag;
APTR TranslatorBase;                    /* Address of translator lib         */

void ErrorText(wrkspc,wrktop)           /* Say error function                */
struct valparam *wrkspc,*wrktop;        /* workspace pointers                */
{
struct valparam *p;                     /* String parameter structure        */
STRPTR outtxt;                          /* Pointer to translated text        */

outtxt = (STRPTR)wrkspc;                /* Point to output buffer            */
p = (--wrkspc);                         /* p = pointer to text               */
```

```
Translate(p->d2,(ULONG)p->d3,outtxt,256);
if ( busy )
  {
   WaitIO(&narrat);
   busy = 0;
  }
narrat.message.io_Command = CMD_WRITE;
if ((narrat.message.io_Length = strlen(outtxt)) != 0 )
  {
   narrat.message.io_Data = (APTR)outtxt;
   SendIO(&narrat);
   busy = 1;
  }
}
int strlen(s)
STRPTR s;
{
  int i;

  for (i=0; *s != '\0'; s++) i++ ;
  return i;
}

                                        /* Signal-routine                  */
void signal(s)
short s;                                /* Signal number as parameter      */
{
  switch(s)
  {
   case 0:
   case 1: break;
   case 2: CleanUp();
   case 3:
   case 4:
   case 5:
   case 6:
   case 7:
   case 8:
   case 9:  break;
   case 10: CleanUp();
  };
}

topptr topnode = NULL;                  /* No procedures or functions      */

void init(pck)
struct package *pck;
{
  SayError = pck;
  pck->var = &topnode;                  /* Initialize top id link          */
  pck->signal = &signal;                /* .. and address of signal routine */

  nar_flag=-1;
  if ((TranslatorBase=OpenLibrary("translator.library",0))==NULL)
     RetStat(199);
```

```
  if ((ReplyPort = CreatePort(0,0)) == NULL)
    {
    CleanUp();
    RetStat(199);
    };
  narrat.message.io_Message.mn_ReplyPort = ReplyPort;
  if (nar_flag = OpenDevice("narrator.device",0,&narrat,0))
    {
    CleanUp();
    RetStat(199);
    };
  pck->Flags |= ErrOutFlag;
  pck->ErrOut = &ErrorText;
  narrat.ch_masks = amaps;
  narrat.nm_masks = 4;
  busy = 0;
}

CleanUp()
{
  if (busy)
   {
   WaitIO(&narrat);
   busy = 0;
   }
  if ( ReplyPort ) DeletePort(ReplyPort);
  if ( nar_flag == 0 ) CloseDevice(&narrat);
  if ( TranslatorBase ) CloseLibrary(TranslatorBase);
  SayError->Flags &= (~ErrOutFlag);
}
```

9. Creating new standard IO devices.

In the previous sections we explained how a package may take over IO from the keyboard and the screen. In many cases it is much more elegant to create a new standard IO device.

In executing a SELECT or an OPEN command AmigaCOMAL searches for the specified unit (file) in a list of standard IO devices, for instance "ds:" (data screen), "kb:" (keyboard) and "sp:" (serial port) and if it is one of these, that device is opened. Otherwise a file is opened.

A device structure is connected to each standard IO device and all these device structures are linked together (I wonder how many structures are floating around inside my Amiga). The device structure contains all the necessary information about the device, i.e. its name, its type and addresses of all its drivers. This is described in more detail in appendix C.4.

A new device may be added to the existing list of devices by calling the AmigaCOMAL routine AddCmlDev (offset -180) with the address of the structure in A0. The new device structure is then put into the front of the chain of device structures.

The device may be removed from the chain by calling the AmigaCOMAL routine RemCmlDev (offset -174) with the address of the structure in register A0.

Example: The following package adds a new output device "con:". This device opens a new window in the WorkBench screen:

```
; Console device package - version 31.12.88

CALL   MACRO
          MOVEM.L  D1/A0-A1/A6,-(A7)
          MOVE.L   \1Base,A6
          JSR      _LVO\2(A6)
          MOVEM.L  (A7)+,D1/A0-A1/A6
       ENDM

       include 'ram:package.i'
       include 'vdk:dos.i'

; Initialization section
```

```
        MOVE.L  D0,A0                   ; Package structure
        MOVE.L  #LinkTop,PckTopId(A0)   ; Set  top link
        MOVE.L  #Signal,PckSignal(A0)   ; .. Signal routine
        MOVE.L  PckCmlStr(A0),A1
        MOVE.L  A1,ComalStr             ; Comal structure
        MOVE.L  DosBaseOff(A1),DosBase
        LEA     ConStruc,A0
        JSR     AddCmlDev(A1)           ; Add new device
        CLR.W   D0                      ; No error
        RTS

; Signal routine
Signal:   CMP.W   #2,D0                 ; DISCARD ?
          BEQ.S   Signal1
          CMP.W   #10,D0                ; BYE ?
          BNE.S   SignalEnd
Signal1:  LEA     ConStruc,A0
          MOVE.L  ComalStr,A1
          JSR     RemCmlDev(A1)         ; Remove device
          MOVE.L  ConHandle,D1
          BEQ.S   SignalEnd
          CALL    Dos,Close
          CLR.L   ConHandle
SignalEnd:RTS

; Open device
Open:     TST.L   ConHandle
          BNE     DevUsed
          MOVEM.L D2-D3/A0/A5,-(A7)
          MOVE.L  D2,A0                 ; Address of device name
          MOVE.L  A5,D1
1$:       MOVE.B  (A0)+,(A5)+           ; Move to workspace
          DBRA    D3,1$
          CLR.B   -(A5)                 ; .. and terminate
          MOVE.L  #1005,D2              ; Mode_old
          CALL    Dos,Open              ; Open window
          MOVEM.L (A7)+,D2-D3/A0/A5
          MOVE.L  D0,D1                 ; Handler to D1
          BEQ     IoErr
          MOVE.L  D1,ConHandle
          CLR.W   D0
          RTS

; Close device
Close:    CALL    Dos,Close
          CLR.L   ConHandle
          CLR.W   D0
          RTS
; Write to device
Write:    EXT.L   D3
          CALL    Dos,Write
          TST.L   D0
          BMI     IoErr
          CLR.W   D0
          RTS
```

```
; Set/get cursor
Cursor:       TST.L   D0
              BPL     SetCursor
              CLR.L   D0                ; No 'Get' implemented
              RTS

SetCursor:    MOVEM.L D0-D3/A0,-(A7)
              LEA     CursorString+1,A0
              MOVE.L  D0,D1
              SWAP    D1
              BSR.S   MakeASCII
              ADDQ.L  #1,A0
              MOVE.W  D0,D1
              BSR.S   MakeASCII
              MOVE.L  ConHandle,D1
              BEQ     SetCursor1
              MOVE.L  #CursorString,D2
              MOVEQ   #7,D3
              CALL    Dos,Write
SetCursor1:   MOVEM.L (A7)+,D0-D3/A0
              RTS

MakeASCII:    EXT.L   D1
              DIVU    #10,D1
              ADD.B   #'0',D1
              MOVE.B  D1,(A0)+
              SWAP    D1
              ADD.B   #'0',D1
              MOVE.B  D1,(A0)+
              RTS

IoErr:        CALL    Dos,IoErr
              RTS

DevUsed:      MOVE.W  #202,D0
              RTS
              SECTION DDATE,DATA

LinkTop:      DC.L    0

; Device structure
ConStruc:     DC.L    0                 ; Room for link
              DC.L    ConName           ; Pointer to name
              DC.W    1                 ; CRT-device
              DC.W    0                 ; Unused
              DC.L    Open
              DC.L    Close
              DC.L    0                 ; No read
              DC.L    Write
              DC.L    Cursor            ; Set/get cursor

ConName:      DC.B    'con:',0
CursorString:   DC.B    $9B,'00;00H'
```

```
      SECTION BLOCK,BSS
;
; Internal workspace
;
ComalStr:     DS.L    1                ; Pointer to Comalstructure
DosBase:      DS.L    1
ConHandle:    DS.L    1

              END
```

Having USEd this package the command

```
SELECT OUTPUT "con:20/10/200/100/ My own window"
```

will direct all output from PRINT statements to this window.

The example shows a special facility. If the name of the device is terminated by a colon (:) you may append supplementary information after the colon while opening the device. This supplementary information will be sent to the open routine of the device (in the example it is sent to AmigaDos without changes).

10. Signals and exceptions.

Packages may allocate signals using the Exec routine AllocSignal and release the signal again using the Exec routine FreeSignal. Further more it may wait for the arrival of a signal using the Exec routine Wait. However, this last call may cause trouble. If the signal does not arrive, it is impossible to break the program.

If you have to wait for a signal it is better to call the AmigaCOMAL routine CmlWait (offset -192). This routine is called in the same way as the Exec routine. But instead of just waiting for the signals specified in the mask it will also wait for all the signals allocated by the AmigaCOMAL system itself.

At return from CmlWait the register D0.L contains the signals received just as the Exec routine Wait. But with CmlWait it may be one of the signals allocated by AmigaCOMAL. It is up to the package to test if it is one of those signals and if it is, the package should call TestBreak to test if the break key has been pressed (see appendix D for further description of TestBreak).

It is possible to make one or more signals cause exception. To obtain this you have to call the AmigaCOMAL routine AddExcept with the address of an exception structure in register A0. This structure must contain a mask for the signals that are to cause exception as well as the address of the exception routine (see appendix C.5). As the result of the call AmigaCOMAL will link the structure into a chain of exception structures (yet another structure!).

At the arrival of one of the signals in the signal mask, the exception routine will be called. This routine should obey the same rules as a normal exception routine.

An exception structure may be removed by calling the routine RemExcept with the address of the structure in A0 (see appendix C.5).

Sometimes it is necessary to synchronize the exception with the execution of an AmigaCOMAL program (for instance if an AmigaCOMAL procedure is called as part of the exception). To achieve this synchronization your exception routine has to set PckExceptFlg (bit 4) in the EventFlgOff field (offset 40) of the comal structure and the PckExceptBit (bit 15) in the PckFlg (offset 10) of the package structure. After the execution of the current line AmigaCOMAL will call the

routine whose address is found in the PckExcept field (offset 52) of the package structure.

11. Communication between packages.

In the field PckUser (offset 64) of the package structure there is room for a user defined pointer.

By making this pointer point to an identification field, it is possible for two or more packages to communicate with each other.

All the package structures are linked together using the first long word in their structure. The address of the first structure in this chain is found in the PckLink field (offset 8) of the comal structure. The last package structure in the chain has a zero in the link field. In searching through the structures you should only search among the initialized machine coded packages, i.e. the packages with type 1 (PckType - offset 8).

Note that although it is up to the user to decide how this pointer is used, you should only place a pointer in the PckUser field, i.e. the field should either contain a zero or the address of something at an even address. Never place the identification word itself in the field!

Appendix A.

Variables in AmigaCOMAL

AmigaCOMAL places three long words on the parameter stack as the actual parameter corresponding to a formal reference parameter. These long words may be picked down from the stack using the following piece of code:

```
MOVE.L  -4(A5),D3           ; Get type
MOVE.L  -8(A5),A3           ; .. address of structure information
MOVE.L  -12(A5),A2          ; .. and address of data field
```

The lower byte (bits 0-7) of the type word contains the type itself while the next byte (bits 8-15) contains the type attributes. For the time being, the high word (bits 16-31) is unused. The possible types are (the symbolic names used are defined in the assembler include file Package.i and in the C include file comal.h):

```
strg (0)          simple text
flt (1)           simple floating point number
long (2)          simple long (32 bits integer)
short (3)         simple short (16 bits integer)
byte (4)          simple byte (8 bits integer)
record (5)        simple record

strgarray (8)     text table
fltarray (9)      floating point table
longarray (10)    long table
shortarray (11)   short table
bytearray (12)    byte table
recordarray (14)  record table
strguser (16)     text function     )
fltuser (17)      float function    )
longuser (18)     long function     )  AmigaCOMAL functions
shortuser (19)    short function    )
byteuser (20)     byte function     )

strgcode (24)     text function     )
fltcode (25)      float function    )
longcode (26)     long function     )  machine coded functions
shortcode (27)    short function    )
bytecode (28)     byte function     )

userproc (128)    AmigaCOMAL procedure
codeproc (129)    machine coded procedure
```

The bits 8-15 (the type attributes) in the type word specify special conditions:

```
POINTERBIT (8)    pointer variable
IMPORTBIT (11)    imported (from package)
CLOSEDBIT (12)    CLOSED
EXTERNALBIT (13)  EXTERNAL
GLOBALBIT (14)    GLOBAL
```

The data fields are constructed in the following way:

```
strg:
    DC.W  maximal length
    DC.W  actual length
    DC.B  the real string content (null terminated)

flt:
    DC.L  mantissa (bit 31 is sign bit)
    DC.W  unused
    DC.W  exponent+$8000

long:
    DC.L  32 bits integer

short:
    DC.W  16 bits integer

byte:
    DC.B  8 bits integer
```

The data fields for the structured variables (arrays and records) are constructed as one or more of the simple variables immediately following each other.

The data field of a pointer variable contains the address of the real data field.

For functions and procedures the data field consists of:

```
    DC.W  number of arguments
    DC.W  type of 1. argument )
    DC.W  type of 2. argument ) for each argument
    DC.W        :             )
    DC.L  routine address (only machine coded)
```

A structure information field is attached to the structured variables (arrays and records). This information field informs about the inner structure of the variable.

The structure information field for an array:

```
    DC.L  length of data field
    DC.W  number of dimensions
```

```
DC.W  lower index for 1. dimension        )
DC.W  number of index values for 1. dim.  )
DC.W  lower index for 2. dimension        ) for each dimension
      :                                   )
      :                                   )
      :
DC.L  address of structure information (only record and array)
```

Structure information field for a record:

```
DC.L  length of data field
DC.L  address of first name field
```

Name fields:

```
DC.L  address of next name field    (0 if last)
DC.B  length of name, ASCII name
DC.W  type of this field
DC.L  offset to data field          (offset from start of record)
DC.L  supplementary information:
                text:               maximal length
                record and array:   address of structure information
```

Appendix B.

Procedure head.

A procedure head contains the necessary information about functions or procedures in a package. In assembler it has the format:

```
DC.L  link (address of next procedure head or 0)
DC.L  address of the name of the procedure (null terminated)
DC.W  type
DC.W  number of arguments
DC.W  type of 1. argument )
DC.W    :                 ) for each argument
DC.W    :                 )
DC.L  address of procedure body (the code itself)
```

For a machine coded function or procedure the possible types are:

```
strgcode (24)
fltcode (25)
longcode (26)
shortcode (27)
bytecode (28)
codeproc (129)
```

The argument types are:

```
strgval   ($001F)
fltval    ($002F)
longval   ($012F)
shortval  ($022F)
byteval   ($032F)
strgref   ($8012)
fltref    ($8022)
longref   ($8122)
shortref  ($8222)
byteref   ($8322)
recref    ($8042)
procref   ($8022)
```

By setting bit 15 in the field containing the number of arguments, an alternative number of argument is specified. This alternative is placed immediately after the routine address and consists of the procedure head starting with the field containing the number of arguments. There is no limitation on the number of alternatives, but they must be listed with increasing number of arguments.

Appendix C.

Data structures in AmigaCOMAL.

As an interface between the packages and the AmigaCOMAL system a number of data structures are used. The content of these structures are listed in the following subsections.

A number of symbolic names are used in the description of the structures. These names are defined in the file Package.i.

The structures are also defined in the C include file comal.h but not all the names used in this file are the same as in the assembler include file. Please consult the listing of comal.h in appendix E.2.

C.1 The package structure.

At the call to the initialization routine of a package AmigaCOMAL places the address of this structure in D0.L.

```
field name   (offset)     content

                (0)       link to next package structure (or 0)
   PckName      (4)       address of package name (null terminated)
   PckType      (8)       the type of the package (set by AmigaCOMAL)
   PckPrior     (9)       the priority of the package (-128 .. 127)
   PckFlag      (10)      the flag of the package
   PckTopId     (12)      address of first  procedure head
   PckSignal    (16)      address of signal routine
   PckCmlStr    (20)      address of comal structure (set by COMAL)
   PckConOut    (24)      address of ConOut routine
   PckConIn     (28)      address of ConIn routine
   PckKbdIn     (32)      address of KbdIn routine
   PckSetCur    (36)      address of SetCur routine
   PckGetCur    (40)      address of GetCur routine
   PckErrOut    (44)      address of ErrOut routine
   PckErrTxt    (48)      address of ErrTxt routine
   PckExcept    (52)      address of Except routine
                (56-63)   reserved
   PckUser      (64)      user defined pointer
```

```
The flag bits are:

   PckExceptBit (15)  call the Except routine of the package
                      bits 14-8:  for the time being unused
   PckErrTxtBit (7)   supplying error texts (through ErrTxt)
   PckErrOutBit (6)   receiving error output (through ErrOut)
   PckGetCurBit (5)   supplying cursor position (through GetCur)
   PckSetCurBit (4)   receiving cursor position (through SetCur)
   PckExclBit   (3)   key$ only from this package
   PckKbdInBit  (2)   supplying key$ (through KbdIn)
   PckConInBit  (1)   supplying inkey$ (through ConIn)
   PckConOutBit (0)   receiving screen output (through ConOut)
```

The type of the package is set by AmigaCOMAL. For the time being the possible values are:

```
   bit 7:       0 = initialized, 1 = uninitialized
   bits 6-0:    0 = AmigaCOMAL,  1 = machine code
```

At the call to the initialization routine of the package AmigaCOMAL has set the link field (offset 0) as well as the fields PckType and PckCmlStr. These fields must remain untouched. The field PckTopId must be set by the package as part of the initialization routine. All the other fields are set to zero by AmigaCOMAL and may be changed by the package at any time.

In the following there is a short description of the routines whose address is found in the package structure.

```
ConOut - output a text on the screen
   at call:     D2.L = address of text
                D3.L = length of text
   at return:   D2.L = address of text
                D3.L = length of text (or zero)
```

If the length of the text is non zero at the return, the text is sent to the next package or to the AmigaCOMAL screen if there is no more packages.

```
ConIn - keyboard input (wait until key pressed)
   at call:     -
   at return:   D0.B = value of key pressed
```

Only one package will be called.

```
KbdIn - keyboard input (return zero if no key pressed)
   at call:     -
   at return:   D0.B = value of key pressed (or zero)
```

If a value of zero is returned the next package will be called. This may be prevented by setting PckExclBit (3) in PckFlag.

```
SetCur - position cursor on screen
   at call:     D0.high = line number
                D0.low = column number
   at return:   D0.L = -1 if no other should position the cursor.

GetCur - return cursor position
   at call:     -
   at return:   D0.high = line number
                D0.low = column number
```

No other packages will be called.

```
ErrOut - show error
   at call:     D0.L = error number
                D2.L = address of error text
                D3.L = length of error text
   at return:   D2.L = address of error text
                D3.L = length of error text (or zero)
```

If the length is non zero at the return, the next package will be called. If there is no other package, the error text will be output on the AmigaCOMAL command screen.

```
ErrTxt - supply error text
   at call:     D0.L = error number
   at return:   D2.L = address of error text
                D3.L = length of error text (or zero)


  Except - call exception routine after execution of COMAL line
    at call:    -
    at return:  -
```

The packages will be called in the order given by the priorities of the packages (highest priority first). If two packages have same priority the one that has been USEd latest will be called first. At the call the registers A5 and A6 are pushed onto the CPU stack and parameters onto the parameter stack (in the order D0,D1,..,A0,..). All registers may be used by the routines.

C.2 Comal structure.

The address of the Comal structure is found in the PckCmlStr field of the package structure. The content is:

```
field name    (offset)     content

   SP_top        (0)       highest stack address
   SP_bot        (4)       lowest stack address
   PckLink       (8)       address of first package in chain of pack.
   TskIdOff      (12)      address of task structure
   DosBaseOff    (16)      DOS library base address
   IntBaseOff    (20)      Intuition library base address
   GraBaseOff    (24)      Graphics library base address
   LayBaseOff    (28)      Layers library base address
   DfBaseOff     (32)      DiskFont library base address (or 0)
   IconBaseOff   (36)      Icon library base address (or 0)
   EventFlgOff   (40)      flags signalling various events
   EventCntOff   (44)      number of unprocessed events
                 (42-55)   reserved
   ComalHdOff    (56)      address of active IO structure
   CommHdOff     (60)      address of command IO structure
   ExecHdOff     (64)      address of execute IO structure
```

The bits in the event flags are:

```
   EscapeFlag (6)      -  BREAK key pressed
   MasterEscFlg (5)    -  quit program
   PckExceptFlg (4)    -  call exception routine of this package
```

C.3 IO structure.

```
 offset        content

  0            address of screen structure
  4            screen type
  6            number of bit planes in screen
  8            screen width (in bits)
 10            screen height (in bits)
 12            address of window structure
 16            number of bit planes in window
 18            window width (in characters)
 20            window height (in lines)
 22            x coordinate of left border of window
 24            y coordinate of upper border of window
 26            width  (in bits)
 28            window height (in bits)
 30            address of font
 34            font height
 36            font width
 38            base line of font
```

```
40          address of virtual window
44          cursor flag
45          actual soft style
46          address of menu structure
50          length of menu structure (in bytes)
```

C.4 Device structure.

A special device structure is connected to each standard device like "ds:", "kb:" and "sp:". The content of this structure is:

```
offset      content

  0         link to next device structure (or 0)
  4         address of device name (null terminated, lower case)
  8         device type
 10         reserved
 12         address of Open routine (or zero)
 16         address of Close routine (or zero)
 20         address of Read routine (or zero)
 24         address of Write routine (or zero)
 28         address of PosDev routine (or zero)
```

The device types are:

```
TypeSeq (0)  -  Sequential device
TypeCRT (1)  -  CRT device
TypeRnd (2)  -  Random device
```

For the five device routines we have:

```
Open - open device
   at call:     D2.L = address of device name
                D3.W = length of device name
   at return:   D0.W = status
                D1.L = open id
```

Normally the device name is the same as the name in the device structure. But if the name in the device structure is terminated by a colon it is possible to add further information after the name (information should be in ASCII).

```
Close - close device
   at call:     D1.L = open id (returned by Open)
   at return:   D0.W = status

Read - Read from device to work space (address in A5)
```

```
   at call:     D1.L = open id (returned by Open)
                D3.L
                   bit 31=1: read until (CR,)LF
                   bit 30-0: maximum number of bytes to read
   at return:   D0.W = status
                A5 points after last byte read

Write - Write to device
   at call:     D1.L = open id (returned by Open)
                D2.L = address of data
                D3.L = length of data
   at return:   D0.W = status

PosDev - set file pointer (only random device)
   at call:     D1.L = open id (returned by Open)
                D2.L = new file position
   at return:   D0.W = status

PosDev - set/return cursor position (only CRT device)
   at call:     D1.L = open id (returned by Open)
                D0.high = new line   )
                D0.low  = new column ) if D0.L > 0
   at return:   D0.L = cursor position, if D0.L < 0 at call
```

Note that no error code is returned.

The routines must save all registers not used to return values. The routines cannot be written in C.

C.5 Exception structure.

offset	content
0	link to next structure
4	mask for signals to cause exception
8	address of interrupt routine

The interrupt routine must satisfy all the same demands as an Exec signal exception routine.

Appendix D.

Calling AmigaCOMAL routines

Below the comal structure there is a jump table which makes it possible to call some useful routines in the AmigaCOMAL system. We will first describe how these routines are called from an assembler programmed package and then how they are called from a C programmed package.

D.1 Calling AmigaCOMAL routines from assembler.

```
name        (offset)     description

CMLRET      (-6)         return
            at call:     D0.W = status code (0 = no error)
                         D2,D3 returned value
            at return:   no return from this routine
```

If the status code is zero the return is to the caller. Otherwise it is an error and the return is to AmigaCOMAL.

```
  NUMADD    (-12)        addition
            at call:     D0,D1 = NUMREG1
                         D2,D3 = NUMREG2
            at return:   D2,D3 = NUMREG2+NUMREG1
                         registers D0-D1/A0-A7 unchanged

  NUMSUB    (-18)        subtraction
            at call:     D0,D1 = NUMREG1
                         D2,D3 = NUMREG2
            at return:   D2,D3 = NUMREG2-NUMREG1
                         registers D0-D1/A0-A7 unchanged

  NUMMUL    (-24)        multiplication
            at call:     D0,D1 = NUMREG1
                         D2,D3 = NUMREG2
            at return:   D2,D3 = NUMREG2*NUMREG1
                         registers D0-D1/A0-A7 unchanged

  NUMDIV    (-30)        division
            at call:     D0,D1 = NUMREG1
                         D2,D3 = NUMREG2
            at return:   D2,D3 = NUMREG2/NUMREG1
```

```
                    registers D0-D1/A0-A7 unchanged

NUMCMP  (-36)       number compare
        at call:    D0,D1 = NUMREG1
                    D2,D3 = NUMREG2
        at return:  flag set as "signed cmp"
                    registers D0-D1/A0-A7 unchanged

FP2INT  (-42)       converts NUMREG2 to integer format
        at call:    D2,D3 = NUMREG2
        at return:  D2,D3 = INT(NUMREG2) as integer
                    registers D0-D1/A0-A7 unchanged

INT2FP  (-48)       converts NUMREG2 to floating point format
        at call:    D2,D3 = NUMREG2
        at return:  D2,D3 = NUMREG2 in float format
                    registers D0-D1/A0-A7 unchanged

NUMSQR  (-54)       square root
        at call:    D2,D3 = NUMREG2
        at return:  D2,D3 = sqr(NUMREG2)
                    registers D0-D1/A0-A7 unchanged

NUMEXP  (-60)       the natural exponential function
        at call:    D2,D3 = NUMREG2
        at return:  D2,D3 = exp(NUMREG2)
                    registers D0-D1/A0-A7 unchanged

NUMLOG  (-66)       the natural logarithmic function
        at call:    D2,D3 = NUMREG2
        at return:  D2,D3 = log(NUMREG2)
                    registers D0-D1/A0-A7 unchanged

NUMPWR  (-72)       power
        at call:    D0,D1 = NUMREG1
                    D2,D3 = NUMREG2
        at return:  D2,D3 = NUMREG2^NUMREG1
                    registers D0-D1/A0-A7 unchanged

NUMSIN  (-78)       sine
        at call:    D2,D3 = NUMREG2
        at return:  D2,D3 = sin(NUMREG2)
                    registers D0-D1/A0-A7 unchanged

NUMCOS  (-84)       cosine
        at call:    D2,D3 = NUMREG2
        at return:  D2,D3 = cos(NUMREG2)
                    registers D0-D1/A0-A7 unchanged

NUMTAN  (-90)       tangent
        at call:    D2,D3 = NUMREG2
        at return:  D2,D3 = tan(NUMREG2)
                    registers D0-D1/A0-A7 unchanged

NUMASN  (-96)       arc sine
```

```
          at call:     D2,D3 = NUMREG2
          at return:   D2,D3 = asn(NUMREG2)
                       registers D0-D1/A0-A7 unchanged

NUMASC    (-102)       arc cosine
          at call:     D2,D3 = NUMREG2
          at return:   D2,D3 = acs(NUMREG2)
                       registers D0-D1/A0-A7 unchanged

NUMATN    (-108)       arc tangent
          at call:     D2,D3 = NUMREG2
          at return:   D2,D3 = atn(NUMREG2)
                       registers D0-D1/A0-A7 unchanged

NUMINT    (-114)       return integer value of NUMREG2
          at call:     D2,D3 = NUMREG2
          at return:   D2,D3 = int(NUMREG2)
                       registers D0-D1/A0-A7 unchanged

CONOUT    (-120)       write text on screen
          at call:     D2.L = address of text
                       D3.W = length of text
          at return:   -
                       registers D0-D7/A0-A7 unchanged

CONIN     (-126)       get value of key pressed (wait if no key)
          at call:     -
          at return:   D0.B = key value
                       registers D1-D7/A0-A7 unchanged

KBDIN     (-132)       scan keyboard
          at call:     -
          at return:   D0.B = key value or zero
                       registers D1-D7/A0-A7 unchanged

SETCUR    (-138)       position cursor
          at call:     D0.high = line number
                       D0.low = column number
          at return:   -
                       registers D0-D7/A0-A7 unchanged

GETCUR    (-144)       return cursor position
          at call:     -
          at return:   D0.high = line number
                       D0.lav = column number
                       registers D1-D7/A0-A7 unchanged

RemExcept (-150)       remove exception routine from exception list
          at call:     A0 = address of exception structure
          at return:   -
                       registers D0-D7/A0-A7 unchanged
```

```
AddExcept  (-156)       add exception routine to exception list
           at call:     A0 = address of exception structure
           at return:   -
                        registers D0-D7/A0-A7 unchanged

TSTSGN     (-162)       test sign of number in NUMEG2
           at call:     D2,D3 = NUMREG2
           at return:   flags set
                        registers D0-D1/A0-A7 unchanged

NUMNEG     (-168)       change sign of NUMREG2
           at call:     D2,D3 = NUMREG2
           at return:   D2,D3 = -NUMREG2
                        registers D0-D1/A0-A7 unchanged

RemCmlDev  (-174)       remove IO device from device list
           at call:     A0 = address of device structure
           at return:   -
                        registers D0-D7/A0-A7 unchanged
```

If streams to the device are open these streams are closed by AmigaCOMAL.

```
AddCmlDev  (-180)       add IO device to device list
           at call:     A0 = address of device structure
           at return:   -
                        registers D0-D7/A0-A7 unchanged

CallUser   (-186)       call Comal procedures/functions
           at call:     D2.L = address of procedure
                        D3.L = type of procedure
                        parameters on parameter stack (A5)
           at return:   returned value in D2,D3 (if any)
                        registers A5-A7 unchanged

CmlWait    (-192)       wait for signals to arrive
           at call:     D0.L = signal mask
           at return:   D0.L = mask for arrived signals
                        registers D1-D7/A0-A7 unchanged
```

The routine waits for the signals in the mask and the AmigaCOMAL signals.

```
TstBreak   (-198)       test for break
           at call:     -
           at return:   D0.B = 0 (not break)
                        $9B (program break)
                        $9A (quit AmigaCOMAL)
                        registers D1-D7/A0-A7 unchanged
```

D.2 Call of AmigaCOMAL routines from C.

The library module comal.lib contains interface routines to the comal routines. The calling conventions are shown below.

```
CmlFlt x,y,z;                  /* CmlFlt defined i comal.h                */
CmlFlt par1, par2, ... ;
struct refparam p;             /* refparam defined in comal.h             */
struct strng *strng;           /* struct strng def. in comal.h            */
struct CmlExcept *CmlExcept;   /* Exception structure                     */
int b;                         /* boolean value                           */
int status;                    /* status value                            */
ULONG cursor;                  /* cursor position                         */
ULONG signal,retsignal;        /* signal masks                            */
   int i;
char c;
APTR wrkbot, wrktop;                 /* bottom and top of work space      */
RetValue(&x);                        /* return value to AmigaCOMAL        */
RetStat(status);                     /* return status to AmigaCOMAL       */
i=FloatInt(x);                       /* convert x to integer              */
x=IntFloat(i);                       /* convert i to comal float          */
y=CmlSqr(x);                         /* y=sqr(x)                          */
y=CmlExp(x);                         /* y=exp(x)                          */
y=CmlLog(x);                         /* y=log(x)                          */
z=CmlPower(x,y);                     /* z=x^y                             */
y=CmlSin(x);                         /* y=sin(x)                          */
y=CmlCos(x);                         /* y=cos(x)                          */
y=CmlTan(x);                         /* y=tan(x)                          */
y=CmlAcs(x);                         /* y=acs(x)                          */
y=CmlAsn(x);                         /* y=asn(x)                          */
y=CmlAtn(x);                         /* y=atn(x)                          */
y=CmlInt(x);                         /* y=int(x)                          */
i=TestSign(x);                       /* i=sign(x)                         */
cursor=CmlGetCusor();                /* return cursor position            */
CmlSetCursor(cursor);                /* position cursor                   */
CmlConOut(strng);                    /* write text on screen              */
c=CmlConIn();                        /* (wait and) get key value          */
c=CmlKbdIn();                        /* return key value or 0             */
c=CmlTstBrk();                       /* test for break                    */
retsignal=CmlWait(signal);           /* wait for signal                   */
AddExcept(CmlExcept);                /* add exception routine             */
RemExcept(CmlExcept);                /* remove exception routine          */
value=CallFunc(f,wrkbot,wrktop);     /* call comal function               */
CallProc(f,wrkbot,wrktop)            /* call comal procedure              */
NumFunc(f,par1,par2, ...)            /* call numeric function             */
```

Note that there are three different routines to call procedures or functions. The last one is used to call numeric functions with numeric parameters only.
During the linking of the package the module comal.lib must be the first library module to be searched in.

Appendix E.

Include-files.

E.1 The assembler include file Package.i.

```
*******************************************************************************
*                                                                             *
*                           AmigaCOMAL                                        *
*                 assembler package include file                              *
*                         version 04.01.89                                    *
*                                                                             *
*******************************************************************************

;  Type attributes
GLOBALBIT:     EQU    14      ; Global variable
EXTERNALBIT:   EQU    13      ; EXTERNAL function/procedure
CLOSEDBIT:     EQU    12      ; CLOED function/procedure
IMPORTBIT:     EQU    11      ; Imported (package)
POINTERBIT:    EQU     8      ; Pointer variable

; Types of variables
strg:          EQU     0      ; Simple string
flt:           EQU     1      ; Simple float
longint:       EQU     2      ; Simple long integer
shortint:      EQU     3      ; Simple short integer
byteint:       EQU     4      ; Simple byte integer
record:        EQU     5      ; Simple record
strgarray:     EQU     8      ; String array
fltarray:      EQU     9      ; Float array
longarray:     EQU    10      ; Long integer array
shortarray:    EQU    11      ; Short integer array
bytearray:     EQU    12      ; Byte integer array
recarray:      EQU    13      ; Record array
strguser:      EQU    16      ; String user FUNCtion
fltuser:       EQU    17      ; Float user FUNCtion
longuser:      EQU    18      ; Long integer user FUNCtion
shortuser:     EQU    19      ; Short integer user FUNCtion
byteuser:      EQU    20      ; Byte integer user FUNCtion
strgcode:      EQU    24      ; String code function
fltcode:       EQU    25      ; Float code function
longcode:      EQU    26      ; Long integer code function
shortcode:     EQU    27      ; Short integer code function
bytecode:      EQU    28      ; Byte integer code function
userproc:      EQU   128      ; User PROCedure
codeproc:      EQU   129      ; Code procedure
```

```
; Types of formal parameters
;
strgval:        EQU     $001F           ; String value
fltval:         EQU     $002F           ; Float value
longval:        EQU     $012F           ; Long integer value
shortval:       EQU     $022F           ; Short integer value
byteval:        EQU     $032F           ; Byte integer value
recval:         EQU     $0042           ; Record value
strgref:        EQU     $8012           ; String REF
fltref:         EQU     $8022           ; Float REF
longref:        EQU     $8122           ; Long integer REF
shortref:       EQU     $8222           ; Short integer REF
byteref:        EQU     $8322           ; Byte integer REF
recref:         EQU     $8042           ; Record REF
procref:        EQU     $8022           ; Procedure REF

; Routine offsets
;
TestBreak:      EQU    -198
CmlWait:        EQU    -192
CallUser:       EQU    -186
AddCmlDev:      EQU    -180
RemCmlDev:      EQU    -174
NUMNEG:         EQU    -168
TSTSGN:         EQU    -162
AddExcept:      EQU    -156
RemExcept:      EQU    -150
GETCUR:         EQU    -144
SETCUR:         EQU    -138
KBDIN:          EQU    -132
CONIN:          EQU    -126
CONOUT:         EQU    -120
NUMINT:         EQU    -114
NUMATN:         EQU    -108
NUMASN:         EQU    -102
NUMACS:         EQU     -96
NUMTAN:         EQU     -90
NUMCOS:         EQU     -84
NUMSIN:         EQU     -78
NUMPWR:         EQU     -72
NUMLOG:         EQU     -66
NUMEXP:         EQU     -60
NUMSQR:         EQU     -54
INT2FP:         EQU     -48
FP2INT:         EQU     -42
NUMCMP:         EQU     -36
NUMDIV:         EQU     -30
NUMMUL:         EQU     -24
NUMSUB:         EQU     -18
NUMADD:         EQU     -12
CMLRET:         EQU      -6
```

```
; Offsets in COMAL structure
;
SP_top:         EQU       0       ; Stack top
SP_bot:         EQU       4       ; Stack bottom
PckLink:        EQU       8       ; Pointer to 1. package
TaskIdOff:      EQU      12       ; Task pointer
DosBaseOff:     EQU      16       ; Dos Base
IntBaseOff:     EQU      20       ; Intuition base
GraBaseOff:     EQU      24       ; Graphics base
LayBaseOff:     EQU      28       ; Layers base
DfBaseOff:      EQU      32       ; Diskfont base
IconBaseOff:    EQU      36       ; Icon base
EventFlgOff:    EQU      40       ; Event flags
EventCntOff:    EQU      41       ; Counter for unprocessed events
ComWinSigOff:   EQU      42       ; IDCMP signal for command window
ComKbdSigOff:   EQU      43       ; Keyboard signal for command window
ExcWinSigOff:   EQU      44       ; IDCMP signal for execute window
ExcKbdSigOff:   EQU      45       ; Keyboard signal for execute window
SerialSigOff:   EQU      46       ; Serial port signal
TimerSigOff:    EQU      47       ; Timer port signal
Res1Off:        EQU      48       ; Reserved for future use
Res2Off:        EQU      52       ; Reserved for future use
ComalHdOff:     EQU      56       ; Pointer to active window str
CommHdOff:      EQU      60       ; Reserved for EditHd
ExecHdOff:      EQU      64       ; Reserved for ExecHd

; Event flag bits
;
EscapeFlag:     EQU       6       ; Break key pressed
MasterEscFlg:   EQU       5       ; Quit program
PckExceptFlg:   EQU       4       ; Call exception routine(s)

; Offsets in package structure
;
PckName:        EQU       4       ; Pointer to package name
PckType:        EQU       8       ; Package type
PckPrior:       EQU       9       ; Priority of package
PckFlag:        EQU      10       ; Flags
PckTopId:       EQU      12       ; Pointer to address of 1. id
PckSignal:      EQU      16       ; Address of signal routine
PckCmlStr:      EQU      20       ; Address of Comal structure
PckConOut:      EQU      24       ; Address of ConOut routine
PckConIn:       EQU      28       ; Address of ConIn routine
PckKbdIn:       EQU      32       ; Address of Keyboard in routine
PckSetCur:      EQU      36       ; Address of SetCursor routine
PckGetCur:      EQU      40       ; Address of GetCursor routine
PckErrOut:      EQU      44       ; Address of Error Output routine
PckErrTxt:      EQU      48       ; Address of Error Text routine
PckExcept:      EQU      52       ; Address of exception routine
PckUser:        EQU      64       ; Offset to user defined pointer
```

```
; Package flag bits
;
PckConOutBit:  EQU      0
PckConInBit:   EQU      1
PckKbdInBit:   EQU      2
PckExclBit:    EQU      3       ; Flag for exclusive KbdIn
PckSetCurBit:  EQU      4
PckGetCurBit:  EQU      5
PckErrOutBit:  EQU      6
PckErrTxtBit:  EQU      7
PckExceptBit:  EQU     15
```

E.2 The C include file comal.h.

```
/* COMAL80 definitions for packages programmed in C       */
/* Version 04.01.89 for Lattice C v. 3.03                 */
/* Lattice C v. 5.xx change the #include to use "         */
/*   example: change  #include <exec/types.h>             */
/*            into    #include "exec/types.h"            */

#ifndef EXEC_TYPES_H
  #include <exec/types.h>
#endif

#ifndef EXEC_TASKS_H
  #include <exec/tasks.h>
#endif

#ifndef INTUITION_INTUITION_H
  #include <intuition/intuition.h>
#endif

/* Parameter types */
#define strgval  0X001f      /* String value parameter                */
#define fltval   0X002f      /* Float value parameter                 */
#define longval  0X012f      /* Long integer value parameter          */
#define shortval 0X022f      /* Short integer value parameter         */
#define byteval  0X032f      /* Byte integer value parameter          */
#define recval   0X0042      /* Record value parameter                */
#define strgref  0X8012      /* String REF parameter                  */
#define fltref   0X8022      /* Float REF parameter                   */
#define longref  0X8122      /* Long integer REF parameter            */
#define shortref 0X8222      /* Short integer REF parameter           */
#define byteref  0X8322      /* Byte integer REF parameter            */
#define recref   0X8042      /* Record REF parameter                  */

/* Identifier types */
#define strg        0        /* String                                */
#define flt         1        /* Float                                 */
#define longint     2        /* Long integer                          */
#define shortint    3        /* Short integer                         */
```

```
#define byteint     4         /* Byte integer                          */
#define record      5         /* Record                                */
#define strgarray   8         /* String array                          */
#define fltarray    9         /* Float array                           */
#define longarray  10         /* Long integer array                    */
#define shortarray 11         /* Short integer array                   */
#define bytearray  12         /* Byte integer array                    */
#define recarray   13         /* Record array                          */
#define strguser   16         /* String user FUNCtion                  */
#define fltuser    17         /* Float user FUNCtion                   */
#define longuser   18         /* Long integer user FUNCtion            */
#define shortuser  19         /* Short integer user FUNCtion           */
#define byteuser   20         /* Byte integer user FUNCtion            */
#define strgcode   24         /* String code function                  */
#define fltcode    25         /* Float code function                   */
#define longcode   26         /* Long integer code function            */
#define shortcode  27         /* Short integer code function           */
#define bytecode   28         /* Byte integer code function            */
#define userproc  128         /* User defined PROCedure                */
#define codeproc  129         /* Code procedure                        */

/* IO-flags          */
#define ConOutFlag   1
#define ConInFlag    1 << 1
#define KbdInFlag    1 << 2
#define ExclFlag     1 << 3
#define SetCurFlag   1 << 4
#define GetCurFlag   1 << 5
#define ErrOutFlag   1 << 6
#define ErrTxtFlag   1 << 7
#define PckIntFlag   1 << 15

/* Type attributes   */
#define POINTERBIT    1
#define IMPORTBIT     8
#define CLOSEDBIT    16
#define EXTERNALBIT  32
#define GLOBALBIT    64

extern APTR WorkTop,WorkBot;

struct OutputStruc                    /* AmigaComal output structure     */
  {
    struct Screen *OutputScreen;      /* Pointer to screen               */
    USHORT ScreenType;                /* Screen type                     */
    USHORT ScreenDepth;               /* Bitmaps in screen               */
    USHORT ScreenWidth;               /* Screen width in bits            */
    USHORT ScreenHeight;              /* Screen height in bits           */
    struct Window *OutputWindow;      /* Pointer to window               */
    USHORT WindowDepth;               /* Bitmaps in window               */
    USHORT LineLen,LineNo;            /* Wd dimensions in char           */
    USHORT GzzXoff,GzzYoff;           /* Gimmezerozero offsets           */
    USHORT WindowWidth;               /* Window width in bits            */
    USHORT WindowHeight;              /* Window height in bits           */
  };
```

```
struct comalstr
  {
    APTR SP_top;                        /* Stack top                    */
    APTR SP_bot;                        /* .. and bottom                */
    APTR PackLink;                      /* Pointer to 1. package        */
    struct Task *task_id;               /* Pointer to Comal task str    */
    APTR DOSbase;                       /* Base address of DOS          */
    struct IntuitionBase *IntBase;
    struct GfxBase *GfxBase;
    APTR LayBase;                       /* layers.library base          */
    APTR DfBase;                        /* diskfont.library base        */
    APTR IconBase;                      /* icon.library base            */
    UBYTE EventFlag;                    /* Flag for events              */
    UBYTE EventCount;                   /* No. of unprocessed events    */
    UBYTE ComWinSig;                    /* IDCMP signal for Com Wind    */
    UBYTE ComKbdSig;                    /* Keyboard signal for Com Wd   */
    UBYTE ExcWinSig;                    /* IDCMP signal for Exec Wd     */
    UBYTE ExcKbdSig;                    /* Keyboard signal for Exec Wd  */
    UBYTE SerialSig;                    /* Serial port signal           */
    UBYTE TimerSig;                     /* Timer port signal            */
    APTR reserv1;                       /* Reserved                     */
    APTR reserv2;                       /* Reserved                     */
    struct OutputStruc *ActStruc;       /* Active output str            */
    struct OutputStruc *CommStruc;      /* Comm. output str             */
    struct OutputStruc *ExecStruc;      /* Exec. output str             */
  };

struct varnode
  {
    struct varnode *node;               /* Pointer to next var-node     */
    STRPTR name;                        /* Pointer to name              */
    UWORD  type;                        /* Type of variable             */
    UWORD  numpar;                      /* Number of parameters         */
  };

typedef struct varnode *topptr;

struct package                          /* Packages structure           */
  {
    struct package *link;               /* Link to next package         */
    STRPTR name;                        /* Pointer to packages name     */
    UBYTE  Type;                        /* Type                         */
    BYTE   Priority;                    /* Priority                     */
    UWORD  Flags;                       /* Flags                        */
    topptr *var;                        /* Init with addr of top-id     */
    void   (*signal)();                 /* Init with addr of signal     */
    struct comalstr *comal;             /* Pointer to comal structure   */
    void   (*ConOut)();                 /* ConOut routine               */
    char   (*ConIn)();                  /* ConIn routine                */
    char   (*KbdIn)();                  /* Keyboard-in  - no waiting    */
    ULONG  (*SetCur)();                 /* Set cursor                   */
    ULONG  (*GetCur)();                 /* Get cursor                   */
    void   (*ErrOut)();                 /* Error output                 */
    void   (*ErrTxt)();                 /* Error text                   */
    void   (*Except)();                 /* Signal exception routine     */
```

```
    APTR   reserved1;
    APTR   reserved2;
    APTR   user;                        /* User defined pointer          */
  };

                                       /*****************************************/
struct CmlExcept                       /* Signal exception structure            */
  {                                    /*                                       */
    struct CmlExcept *link;            /* Link to next structure                */
    ULONG  Signal;                     /* Signal mask                           */
    void   (*SigExcept)();             /* Exception routine                     */
  };                                   /*****************************************/

                                       /*****************************************/
struct strng                           /* String format in COMAL                */
  {                                    /*                                       */
    short max;                         /* maximal length field                  */
    short akt;                         /* actual length field                   */
    char txt[2];                       /* string content                        */
  };                                   /*****************************************/

                                       /*****************************************/
struct refparam                        /* REF parameter format                  */
  {                                    /*                                       */
    APTR  Data;                        /* Address of data                       */
    APTR  Information;                 /* Information block address             */
    UWORD dummy;                       /* Register D3.high - not used           */
    UBYTE Attributes;                  /* Type attributes                       */
    UBYTE Type;                        /* Type                                  */
  };                                   /*****************************************/

                                       /*****************************************/
struct ptrparam                        /* Pointer parameter                     */
  {                                    /*                                       */
    APTR  *Data;                       /* Pointer to address of data            */
    APTR  Information;                 /* Information block address             */
    UWORD dummy;                       /* Register D3.high - not used           */
    UBYTE Attributes;                  /* Type attributes                       */
    UBYTE Type;                        /* Type                                  */
  };                                   /*****************************************/

                                       /*****************************************/
struct valparam                        /* Value parameter format                */
  {                                    /*                                       */
    long  d2;                          /* register D2.L                         */
    long  d3;                          /* register D3.L                         */
  };                                   /*****************************************/

typedef double CmlFlt;
```

Appendix F.

References.

There are a lot of books describing the 68000 assembler on the market. Here we will list a single title:

Kane, Hawkins, Leventhal
68000 Assembly Language Programming
OSBORNE/McGraw-Hill

The C programming language is also described in a lot of books.

If you intend to use the facilities of the Amiga it is necessary to get some books describing the machine. Fortunately there are a lot of such books. Start with the four books made by Commodore:

Amiga ROM Kernal Reference manual: Exec
Amiga ROM Kernal Reference manual: Libraries and Devices
Amiga Intuition Reference Manual
Amiga Hardware Reference Manual

Among the other books we will mention:

Eugene P Mortimore
Programmers Handbook volume 1+2
SYBEX

NOTES

NOTES

NOTES