AMIGA-COMAL Version 3.4

{**Author: Svend@DaugaardPedersen.dk (Svend Daugaard Pedersen**)
{Guide modified/adapted to MS Word by Misel Zivanovic in 2025, rev. 1.0.0}

I. GETTING STARTED

1 Hardware requierements.

The Comal programming system will run on any Amiga computer having at least 1 megabyte of main memory and one diskette drive.

Although one disk drive will suffice an extra disk drive (or a hard disk) is recommended.

2 Software requirements.

Comal requires Workbench 1.3 or higher to run.

To be able to use the Comal Intuition Tool (CIT) Workbench 2.04 or higher is needed.

3 Creating a backup disk.

Comal is supplied on two 3.5" floppy disks named Comal and Comal.Extras. Before installing Comal, make a backup copy of the original disks and store these in a safe place.

Before doing this make sure that the disks are write protected (you should be able to look through the small hole in the plastic disk cover).

If your Amiga has only one disk drive the backup is done in this way:

1. Put the write protected Comal (Comal.Extras) disk into a disk drive.

2. Select the Comal (Comal.Extras) disk by clicking once on its icon with the left mouse button.

3. Choose the Copy item from the Icons menu (in WB1.3 the duplicate item from the Workbench menu) and follow the instructions in the requesters.

4. After the copy has been made, choose the Rename item from the Icons menu (the Workbench menu in WB1.3) and remove the "copy_of_" from the name.

If your Amiga has two disk drives the backup is done in this way:

1. Put the write protected Comal (Comal.Extras) disk into one disk drive.

2. Write enable a blank floppy disk and put it into the other disk drive.

3. Select the Comal (Comal.Extras) disk by clicking once on its icon with the left mouse button and drag it on top of the icon of the blank disk.

4. After the copy, remove the original Comal (Comal.Extras) disk from the disk drive, select the "copy_of_Comal" ("copy_of_Comal.Extras") disk by clicking on its icon with the left mouse button and choose the Rename item from the Icons menu (the Workbench menu in WB1.3)and remove the "copy_of_" from the name.

4 Installing Comal on a floppy disk system.

The disk named Comal is ready to use. Put the copy of this disk into a disk drive and open the disk by double clicking on its icon.

5 Installing Comal on a hard disk system.

Put the copy of your Comal.Extras disk into a disk drive and open the disk by double clicking on its icon.

Make an empty drawer on your hard disk by using the New drawer item in the Window menu (use the empty drawer in WB1.3) and give the drawer the name Comal (or another name). Select InstallHD by clicking on its icon with the left mouse button. Press the shift key and double click on the icon of the newly created drawer.

As an alternative to creating a new drawer your self you may just start the InstallHD program by double clicking on its icon. Then you will be prompted for the name of the destination drawer (that will be created if not found).

During the installation you will be asked to put in the other disk and you will be asked if the ARexx demo scripts should be copied to the Rexx: directory (if ARexx is installed).

The InstallHD program does not do anything else than copying needed files to the selected drawer and (if you accepted) ARexx scripts to Rexx:.

6 Starting Comal from Workbench.

Comal may be started from Workbench in the normal Amiga way by double clicking on the Comal icon.

If you are using a one drive Amiga system you will be asked to change disk once during the loading (Comal has to load some disk based libraries). Normally this is only the first time you are starting Comal.
The next time the libraries are already loaded.

In stead of double clicking on the Comal icon you may click on the icon of a Comal program text (an ASCII text containing a Comal program). By doing this Comal will be started and the program will be loaded into the editor.

7 Starting Comal from the Shell (the CLI).

Comal may be started from the Shell by executing a command of the format:

```
Comal [startup parameters] [Program]
```

where Program is the name of a Comal program eventually preceded by the path to the directory containing the program and the optional startup parameters are of the same form as the tool types (se section 8.1).

If you are specifying a program name, Comal will be started and the program will be loaded. The current directory will be the directory containing the program.

Example:

```
Comal CITDemos/TinyPaint      (Start Comal and load CITDemos/TinyPaint)

Comal WORKSPACE=100000        (Start Comal with 100000 BYTES workspace)
```

8 Setting the Comal environment.

Several parameters used by the Comal programming environment may be set either by using the standard Workbench info requester or by using the Prefs program found in the Prefs drawer.

8.1 Setting the Comal TOOLTYPES.

The info requester can be displayed by selecting the Comal icon and then select the Information... item from the Icons menu in Workbench (WB1.3: select the Info item in the Workbench menu).

The field Tool Types inside the Info requester is where you can set the Comal parameters. Each of the parameter are set by using one line of the format:

```
TOOLTYPE=ToolValue
```

The TOOLTYPE's and the ToolValue's you can use are described below.

SCREEN

By using the SCREEN tool type you select if Comal should open its own screen or use the workbench screen. The possible tool values are the name of a public screen to be used (if the screen is not found it will be created).

Example: SCREEN=Comal

The only legal value for WB1.3 users is Workbench.

The default value is a private screen.

DEPTH

By using the DEPTH tool type you select depth of the screen. The default value is 2.

Example: DEPTH=4

LANGUAGE

By using this tool type you may select the language of the texts used in the Comal system.

Example: LANGUAGE=dansk

The default language is English.

WINDOW

By using this tool type you select the position and size of the editor window at start. The format is

```
WINDOW=LeftEdge,TopEdge,Width,Height
```

Example: WINDOW=95,30,540,200

ENTERMODE

Selects the effect of the enter key. The tool values are Down and Insert. If you choose Insert the cursor is moved down and a new, empty line is inserted each time the ENTER key is pressed. If you choose Down the cursor is moved down but no new line is inserted. The default value is Insert.

Example: ENTERMODE=Down

The behavior of the ENTER key may at any time be changed inside Comal.

KEYWORD

Selects the letters used to list keywords. The tool values are Capital and Small. The default value is Capital.

Example: KEYWORD=Small

This parameter may at any time be changed inside Comal.

BACKUP

Selects the backup option. The tool values are On and Off. If on a backup file is created each time a program text is stored on disk. The default value is On.

Example: BACKUP=Off

This parameter may at any time be changed inside Comal.

ICON

Selects the icon option. The tool values are On and Off. If on an icon is created when a program is stored on disk. If the program is stored as a text file, this icon can be used to start Comal and load the program. If it is stored as a code file, the icon can be used to start execution of the program (without loading the editor). The default value is On.

Example: ICON=Off

This parameter may at any time be changed inside Comal.

LINEEND

Selects the lines are terminated in text files. The tool values are LF, CR, CR+LF and PC. The default value is LF (standard Amiga).

Example: LINEEND=CR+LF

This parameter may at any time be changed inside Comal.

AUTOVAR

This tool type is used to select if a running program should create variables automatically or if all variables has to be declared in a DIM or LOCAL statement before use. The default value is On.

Example: AUTOVAR=Off

This parameter may at any time be changed inside Comal.

EXECWINDOW

This tool type is used to select if a running program should open the standard execute window (used by PRINT, INPUT etc.). The default value is On.

Example: EXECWINDOW=Off

This parameter may at any time be changed inside Comal.

PROGRAM

Selects the size of the program buffer. The default value is 25000 bytes.

Example: PROGRAM=20000

WORKSPACE

Selects the size of the workspace used by a running program. The default value is 75000 bytes.

Example: WORKSPACE=60000

STACK

Selects the size of the stack for a running program. The default value is 8Kb.

Example: STACK=16000

MACRO

By using this tool type you select the name of the file that contains the macro definitions that will be assigned to the function keys. The default file is Comal.Macro.

Example: MACRO=Comal:MyMacro

The tool types (except the SCREEN and DEPTH tool types) can be used in icons for program texts. If this is done, only the parameters for that project is affected. If another project is started (see section 3.1) the standard parameters or the values set in the Comal icon is used.

The parameter values put into the Comal icon is read even if Comal is started from the Shell.

8.2 The Preferences program Pref.

Most of the parameters that can be set by using the TOOLTYPES in the Comal info requester can be set in a much more comfortable way by running the program Prefs in the Prefs drawer.

NOTE:
Workbench 1.3 users can only set the language in this way.

II. TUTORIAL

This chapter is intented to help beginners through the first difficult steps in making their first programs. You will learn to make simple programs, to use some of the menus in Comal and to debug your program.

Skilled programmers may skip this chapter or just skim it.

1 Your first Comal programs.

Having started Comal by clicking on its icon you will se a window - the edit window. This window is used to type in your programs.

Start by entering this program consisting of only one line:

```
print "Hello there!"
```

and press <ENTER>. Nothing happens except that the word print is rewritten in capital.

Now, let's try to execute it. Press the mouse menu button and select the Execute item in the Program menu (or use the alternative short form of this menu selection: <right Amiga>+<E>). As a result a new window is opened (the output window) and the text Hello there! is printed in this window. After that the edit window is brought to front again.

Let's remove this program and make a little longer program. Select the Clear Program Buffer menu item from the Project menu. As a result a requester will pop up. This is because the program you just made has not been saved on disk. You will probably not want to save this little program, so just click on Yes. The edit window will now be cleared and the execute window will disappear.

Enter this program:

```
INPUT AT 10,5: "Type in your name: ": Name$

PAGE  // Erase window

PRINT AT 10,20: "Hello ",Name$,"!"
PRINT
PRINT AT 12,20: "Welcome to Comal."
```

Remember to press <ENTER> after each line. Be careful to type in excactly the same as shown. If you do not the Comal system will probably ask you to correct the line.

Let's execute this program. Select the Execute menu item from the Program menu (or use the short form: <right Amiga>+<E>). As a result the execute window will again be opened and you are asked to type in your name. Do that and terminate by pressing <ENTER>. The window will be cleared and a text will be printed.

If you did not typed excactly as shown the program may stop, an error message will be printed in the status line of the edit window and the cursor will be positioned on (or just after) the place where the error was detected. Correct the error and execute the program again.

If the program terminates without error you will notice that the edit window will be brought to front, so that you cannot see the text. This is bad.

Use the cursor keys or the mouse to move the cursor to the end of the program and add this line:

```
WAIT 3  // Wait 3 secs
```

Run the program again (<right Amiga>+<E>). The program waits 3 seconds before it stops. Then you have the time to see the output.

2 Working with graphics.

In this and the following sections we will make programs that draws small figures in the window. But before we continue, you should clear the program buffer. Use the the Clear Program Buffer menu item in the Project menu.

Comal is an interactive programming language. This means that you can execute commands immediately without putting them into a program. We will use this facility to show you how some of the drawing routines works.

Select the Open Command Window item from the Project menu (or use the short form: <right Amiga>+<K>). As a result a little window (the command window) will appear in the top left corner of your screen. In this window some of the Comal statements may be executed as commands. Try for instance to write this line:

```
PRINT 2+3
```

and press <ENTER>. The expression will be calculated and the result (5) will be printed at once.

To be able to use the graphics routine you have to link one of the graphics modules Graphics and Turtle into the system. We will choose Turtle.

The linking is done by executing the line:

```
USE Turtle
```

Type this line into the command window and terminate by pressing <ENTER>. Comal reads the module from disk and makes all the routines within this module accessible.

Before you can make drawings, you have to open the graphics system. This is done by writing:

```
graphicscreen(0)
```

(remember to press <ENTER>).

If you have typed in commands as above the execute window will be opened and a small triangle (a turtle) will be drawn in the middle of the window.

Now we are ready to use the graphics routines in the Turtle module. Let's move the turtle forward. This is done by writing the line

```
forward(50)
```

(remember <ENTER>). The result is that the turtle is moved 50 units forward and a line is drawn behind it.

The turtle may be turned in any direction. This is done by writing the line right(90)

which turns the turtle 90o to the right.

If you repeat these two commands three more times (each repetition is most easily done by moving the cursor up two lines and then pressing <ENTER> twice) a nice square will be drawn.

In the Turtle module there are a lot of different graphics routines. Read more about these in the description of the graphics modules.

Before we leave this section you should close the graphics system by writing the line:

```
textscreen
```

and then close the command window by clicking on its close gadget in the upper left corner of the window.

3 Repetition.

The drawing routines used in the preceeding section may be used as program statements as well. Here is a program that draws the square from that section:

```
// Program 3.1

USE Turtle

graphicscreen(0)  // Open the graphics system

forward(50)       // Draw a square
right(90)
forward(50)
right(90)
forward(50)
right(90)
forward(50)
right(90)

WAIT 4            // Time to look at the drawing

textscreen        // Close the graphics system
```

You may type in the program in the editor or load it from disk where it is stored. Select the Open... menu item in the Project menu (or use the short form: <right Amiga>+<O>). A file requester will open.

Click on the scroll bar in the right side of the requester and move it to the bottom. Then click on the name Tutorial which will be written in the drawer field of the requester (alternatively you may just write the name Tutorial in this field). Now select the file Prg3.1 and then press the OK gadget. As a result the program will be loaded.

Try to execute the program (<right Amiga>+<E>). It should draw a nice square in the execute window. The program will wait four seconds before the edit window will be brought to front.

In the program, the two statements:

```
forward(50)
right(90)
```

are repeated four times. The program can be made more elegant by making the Comal system automatically repeat the two statements. This can be done by using the LOOP statement:

```
LOOP 4 TIMES
    :
  statements to be repeated
    :
ENDLOOP
```

Let's try to modify the program. Move the cursor to line number 9 (use the cursor keys or the mouse). Select the Mark Block Start menu item in the Edit menu (or short: <right Amiga>+<B>). The editor will change to block mode (which is written in the status line) and the cursor line (line 9) is written in blue (red in WB 1.3). Move the cursor down to line number 14 using the cursor key (the mouse cannot be used in block mode). As seen the lines 9-14 are marked as a block.

Now delete this block by selecting the Erase menu item. As a result a requester will pop up and you are requested to confirm. Press the Yes gadget to accept the operation.

Move the cursor three lines up (to line 6) and press <ENTER> to insert an extra line. Write the line

```
LOOP 4 TIMES
```

and don't press <ENTER>. If you did press <ENTER> an extra line has been inserted. Delete this by pressing <shift>+<Del>. Move the cursor down three lines (line 10 just after right(90)). Note that the lines are indented while you are moving down.

Write the line

```
ENDLOOP
```

and press <ENTER> to insert an extra line.

May be you should change the program header line (line 1). Move the cursor to the end of this line and delete the last 1 by pressing back space (the key on the left side of the Del key above the <ENTER> key). Change the line to:

```
// Program 3.2
```

The changed program should now look like:

```
// Program 3.2

USE Turtle

graphicscreen(0)  // Open the graphics system

LOOP 4 TIMES
  forward(50)     // Draw a square
  right(90)
ENDLOOP

WAIT 4            // Time to look at the drawing

textscreen        // Close the graphics system
```

Try to execute the program to make sure that it works just as before.

Save the program on the RAM: disk (just as an exercise). Select the Save As... menu item in the Project menu (short form: <right Amiga>+<A>). The file requester will open. Select Disks and RAM: and write the name Prg3.2 in the File text field of the requester. Then save by clicking on the OK gadget (or by pressing <ENTER>).

By using the LOOP statement it is easy to make the program draw other figures. Change the LOOP section to

```
LOOP 3 TIMES
  forward(50)
  right(120)
ENDLOOP
```

to draw a triangle, or to

```
LOOP 6 TIMES
  forward(50)
  right(60)
ENDLOOP
```

to draw a hexagon. Or try this funny little figure by changing it to

```
LOOP 20 TIMES
  forward(50)
  right(168)
ENDLOOP
```

Comal contains a number of other repetition statements: WHILE, REPEAT and FOR statements. Se chapter IV for a complete description.

4 Selection.

In section 3 we learned to draw triangles, squares etc. In this chapter we will make a program that can draw any of these figures dependent of the users choice.

A skeleton for such a program would be:

```
Program figures
  initialize
  print the possible choices
  get users choice
  draw the chosen figure
```

The initialization is simply:

```
graphicscreen(0)
```

The printing section could be made in this way:

```
PRINT AT 6,10: "Triangle ........ 3"
PRINT AT 8,10: "Square .......... 4"
PRINT AT 10,10: "Pentagon ........ 5"
PRINT AT 12,10: "Hexagon ......... 6"
```

The users choice is made by a single INPUT statement:

```
INPUT AT 16,10: "Select figure: ": choice
```

To draw the correct figure we have to use of one of the selection statements in Comal. Here we will use the IF statement:

```
IF choice=3 THEN
  draw triangle
ELIF choice=4 THEN
  draw square
ELIF choice=5 THEN
  draw pentagon
ELIF choice=6 THEN
  draw hexagon
ELSE
  print error
ENDIF
```

The complete program looks like:

```
// Program 4.1

USE Turtle

graphicscreen(0)
ht

PRINT AT 6,10: "Triangle ......... 3"
PRINT AT 8,10: "Square ........... 4"
PRINT AT 10,10: "Pentagon ......... 5"
PRINT AT 12,10: "Hexagon .......... 6"

INPUT AT 16,10: "Select figure: ": choice

page

IF choice=3 THEN    // Triangle
  LOOP 3 TIMES
    forward(50)
    right(120)
  ENDLOOP
ELIF choice=4 THEN  // Square
  LOOP 4 TIMES
    forward(50)
    right(90)
  ENDLOOP
ELIF choice=5 THEN  // Pentagon
  LOOP 5 TIMES
    forward(50)
    right(72)
  ENDLOOP
ELIF choice=6 THEN  // Hexagon
  LOOP 6 TIMES
    forward(50)
    right(60)
  ENDLOOP
ELSE
  PRINT AT 12,10: "I don't know that figure!"
ENDIF

WAIT 4
textscreen
```

You may read this program from the disk (<right Amiga>+<O>). The name of the program is Prg4.1). Execute it. Remember that a selection of a number must be terminated by pressing <ENTER>.

5 Procedures.

The program in section 4 would be more readable if we could write the selection part in this way:

```
IF choice=3 THEN
  triangle
ELIF choice=4 THEN
  square
ELIF choice=5 THEN
  pentagon
ELIF choice=6 THEN
  hexagon
ELS
  PRINT AT 12,10: "I don't know that figure!"
ENDIF
```

To do this we have to define procedures for each of the figures in question. The square procedure for instance would be:

```
PROC square
  LOOP 4 TIMES
    forward(50)
    right(90)
  ENDLOOP
ENDPROC square
```

Let's try to change the program in this way.

Load the program Prg4.1 into the editor (if it is not already there) and move the cursor to line 17 (the first LOOP statement). Select the Mark Block Start menu item in the Edit menu (or short: <right Amiga>+<B>) and move the cursor three lines down (so that the whole LOOP-ENDLOOP structure is marked as a block).

Cut the block off the program and move it into the clip board by selecting the Cut menu item in the Edit menu (short: <right Amiga>+<X>). Move one line up and press <ENTER> to make an empty line and write triangle in this line (this line works as a replacement for the removed block).

Move to the end of the program by pressing <Ctrl>+<down arrow>, make an extra empty line by pressing <ENTER> and write

```
PROC triangle
```

and press <ENTER> to move to the next line.

Insert the block from the clip board by selecting the Paste menu item in the Edit menu (short: <right Amiga>+<V>) and move to the line after ENDLOOP and write:

```
  ENDPROC triangle
```

Make similar changes for the other figures. The result should look like:

```
  // Program 5.1

  USE Turtle

  graphicscreen(0)
  ht

  PRINT AT 6,10: "Triangle ......... 3"
  PRINT AT 8,10: "Square ........... 4"
  PRINT AT 10,10: "Pentagon ......... 5"
  PRINT AT 12,10: "Hexagon .......... 6"
  INPUT AT 16,10: "Select figure: ": choice

  PAGE
  IF choice=3 THEN    // Triangle
    triangle
  ELIF choice=4 THEN  // Square
    square
  ELIF choice=5 THEN  // Pentagon
    pentagon
  ELIF choice=6 THEN  // Hexagon
    hexagon
  ELSE
    PRINT AT 12,10: "I don't know that figure!"
  ENDIF

  WAIT 4
  textscreen

  // ********* end of main program **************

  PROC triangle
    LOOP 3 TIMES
      forward(50)
      right(120)
    ENDLOOP
  ENDPROC triangle

  PROC square
    LOOP 4 TIMES
      forward(50)
      right(90)
```

```
    ENDLOOP
  ENDPROC square

  PROC pentagon
    LOOP 5 TIMES
      forward(50)
      right(72)
    ENDLOOP
  ENDPROC pentagon

  PROC hexagon
    LOOP 6 TIMES
      forward(50)
      right(60)
    ENDLOOP
  ENDPROC hexagon
```

The program is not shorter neither is it faster, but it is much more readable, since you only need to read the first 30 lines to see what it does (provided you have used descriptive names).

The four procedures at the end of the program contain almost the same code. We could use this fact to make the program shorter. A general Polygon procedure could be made in this way:

```
  PROC Polygon(n)
    LOOP n TIMES
      forward(50)
      right(360/n)
    ENDLOOP
  ENDPROC Polygon
```

By using this, the four procedures should be changed to (note that the new procedure must be written Polygon and not polygon or you will conflict with one of the procedures in the Turtle module):

```
  PROC triangle
    Polygon(3)
  ENDPROC triangle

  PROC square
    Polygon(4)
  ENDPROC square

  PROC pentagon
    Polygon(5)
  ENDPROC pentagon
```

```
PROC hexagon
  Polygon(6)
ENDPROC hexagon
```

Try to make these changes as an exercise. The changed version is stored on disk under the name Prg5.2. Execute the program to see if it still does the same.

One of the advantages of using procedures is that the program is more readable. But it is also much easier to make changes in a program that uses procedures. Try for instance to type in the following little procedure at the end of your program:

```
PROC Fwrd(length)
  IF length<10 THEN
    forward(length)
  ELSE
    Fwrd(length/3)
    left(60)
    Fwrd(length/3)
    right(120)
    Fwrd(length/3)
    left(60)
    Fwrd(length/3)
  ENDIF
ENDPROC Fwrd
```

and replace the line forward(50) in the polygon procedure (line 51 in the program Prg5.2) with the line

```
Fwrd(50)
```

Execute the program to see the effect of the changes. It is very surprising!

6 Making modules.

As you saw in the last section it is a good idea to hide "complicated" code in procedures and put these procedures in a separate section of the program. Another good reason for using procedures is that the same piece of code may be reused. The Polygon procedure is a good example of this (it was used four times).

The idea of hiding and reusing code may be developped further by putting the procedures into a module and store this module on disk. This is in fact what has been done in the Turtle module. In this section it will be shown how such a module is made. As an example we will put the four procedures triangle, square, pentagon and hexagon into a module called Figures so that they can be used in other programs.

First you have to load the program Prg5.2 into the editor (<rightAmiga> +<O>). Then go to the line before the first procedure (line 32). Press <ENTER> to insert an extra line and write the following in this empty line:

```
MODULE Figures
```

Go to the end of the program (note that the lines are being indented as you are moving down), insert an extra line and write:

```
ENDMODULE Figures
```

Now you have made an internal module which can be tested before it is stored on disk. But the module is not ready for test. You have to USE the Turtle module inside this module and you have to tell which ones of the procedures should be made accessible from outside. This is done in special EXPORT statements. The final module should look like this:

```
MODULE Figures

  USE Turtle

  EXPORT triangle,square,pentagon,hexagon

  PROC triangle
    Polygon(3)
  ENDPROC triangle

  PROC square
    Polygon(4)
  ENDPROC square

  PROC pentagon
    Polygon(5)
  ENDPROC pentagon

  PROC hexagon
    Polygon(6)
  ENDPROC hexagon

  PROC Polygon(n)
    LOOP n TIMES
      forward(50)
      right(360/n)
    ENDLOOP
  ENDPROC Polygon

ENDMODULE Figures
```

Put the USE statement and the EXPORT statement into the module so that it looks as above.

Now let's try to execute the program. Press <right Amiga>+<E> and select 3 (triangle) and press <ENTER>.

What happened? An error occured! The cursor is placed after the word triangle and an error message in the status line tells you that the name is unknown. This is because you have not told the main program to import the procedures from the module. You have to add the line

```
USE Figures
```

in the start of the program (for instance just after the line: USE Turtle). The start of the main program should look like:

```
// Program 6.1

USE Turtle
USE Figures

graphicscreen(0)
```

Having added the USE line to your program, you are ready to test the module once more. This time it should be okay. The program is stored on disk under the name Prg6.1

The next step in making a module is to store it on disk.

First you have cut the module off the program and move it into the clip board. Go to the MODULE line (line 34 of Prg6.1) and enter block mode (<right Amiga>+<B>). Move to the ENDMODULE line (line 63). Now the whole module is marked as a block. Cut it off the program (<right Amiga>+<X>).

As the next step you have to open a new project. Select the New menu item in the Project menu (<right Amiga>+<N>). As a result a new editor window will be opened. It is named Project 2. In fact you have started a new process in the Amiga. In this new project you can edit and execute programs independent of what you are doing in the original project. And you can even run programs in the two projects in parallel. But we are not going to do that now.

The clip board we have used is the standard Amiga clip board so that you can get the content from the newly started project (and other programs that uses this clip board). Select the Paste menu item in the Edit menu (<right Amiga>+<V>) and the module will be copied into your editor.

And now we have to store it on disk. But in contradiction to what we have done earlier, where we have stored programs on disk as simple text files (ready to be loaded by any text editor like for instance MEmacs or ED), we have to store it in a special code form.

Select the Save... menu item in the Program menu (no short form - sorry!). As a result the well known file requester will open. Activate the Drawer text field by clicking on it with your mouse (the left mouse button) and write

```
Comal:Modules
```

Then write the following name into the File text field:

```
Figures.mod
```

Be careful to write it correct! Press <ENTER> (or click on the OK gadget) to store the module.

That's all. Close Project 2 by clicking on the close gadget and start your program in Project 1. You will notice that it starts reading the module from disk. But otherwise the program works as before.

Store your program on disk in the drawer Tutorial under the name Prg6.2 (<right Amiga>+<A>).

7 Looking into modules.

If you are using externale modules stored on disk it may happen that you want to see the content of a specific module. If there is a written documentation for that module, you may find the answer to your questions in this. But if you do not have such a documentation there is another possibility.

Load the program Prg6.2 (if it is not already in the editor) and execute it so that modules are loaded. Select the Show Modules... menu item in the Program menu (short: <right Amiga>+<H>). As a result a file requester like window will pop up showing you all modules loaded so far.

You may wonder! Five modules are shown but you have only loaded the Turtle module and the Figures module. The explanation is easy. The Turtle module itself uses the Graphics module and the System module which itself uses the SystemCode module.

Select the Figures module either by pressing <ENTER> or by clicking on the Accept gadget. As a result your Figures module is shown in the editor window. The status line tells you that you are in module mode and what the

name of the module is. In module mode you can move around in the text but you cannot edit the text and a lot of functions are disabled.

To go back to the main program select the Edit Main Program menu item in the Edit menu (short: <right Amiga>+<M>).

Let's see the loaded modules again (<right Amiga>+<H>).

Modules may be written in Comal or they may be mashine coded modules. The Figures, Turtle and System modules are Comal modules while Graphics and SystemCode are mashine coded modules. Select the Graphics module (for instance by double clicking on the name).

This time a new window is opened. The source of the module is not shown, but an interface containing all the procedures and functions are shown. Move around in the text by using the cursor keys or by using the scroll bars. Click on the close gadget in the top left corner of the window to remove the window.

Try to look into the other modules, for instance the Turtle module. It is a large and rather complicated module that will convince you of the advan tage of hiding code in modules.

8 Making gadgets.

The programs in sections 4-6 used the INPUT statement to get a users choice. This is somewhat old fashioned. A better and more Amiga like method is to use gadgets so that the user can press on the relevant gadget by using the mouse.

By using the CIT-modules (CIT is short for Comal Intuition Tool) it is relatively easy to make programs with gadgets. NOTE! CIT can only be used with WB2.04 or higher.

Load the program Prg6.1 or Prg6.2 (if not already in the editor). First add two USE lines specifying the CIT-modules we are going to use:

CITWindows and CITGadgets.

The start of the program should look like this:

```
// Program 6.1

USE CITWindow
USE CITGadgets
USE Turtle
USE Figures

graphicscreen(0)
ht
```

First we'll make a gadget for the triangle. This is done by executing a DIM statement of the form:

```
DIM TriangleGad OF ButtonGadget
```

The gadget is a simple ButtonGadget. By using CIT a lot of other gadgets can be made, but in this section only button gadgets will be used.

We have to tell CIT what the size of the gadget should be and where the gadget is to be placed. This is done in the following statements:

```
TriangleGad.Size(120,20)    // Width = 120 and height = 20 pixels
TriangleGad.Position(0,0)   // The upper left corner of the window
```

The point between TriangleGad and Size/Position indicates that Size and Position are in fact methods in a data structure. Read chapter IV.8 if you want to know more about methods and objects. But here you do not need think more about that (just write as shown).

Inside the gadget we want the label Triangle.
This is specified in this way:

```
TriangleGad.Label("Triangle",INSIDE)
```

Now we have specified the layout and the position of the gadget. It is inserted in the standard Comal window in this way:

```
ComalWindow.InsObject(TriangleGad,Error)
```

As a result of executing this statement the gadget will be visible in the window. If an error occurs an error message is returned in the variable Error. We will not test for errors until all gadgets are made.

The gadgets for the other figures are made in the same way:

```
DIM SquareGad OF ButtonGadget
SquareGad.Size(120,20)
SquareGad.Position(120,0)
SquareGad.Label("Square",INSIDE)
ComalWindow.InsObject(SquareGad,Error)

DIM PentagonGad OF ButtonGadget
PentagonGad.Size(120,20)
PentagonGad.Position(240,0)
PentagonGad.Label("Pentagon",INSIDE)
ComalWindow.InsObject(PentagonGad,Error)

DIM HexagonGad OF ButtonGadget
HexagonGad.Size(120,20)
HexagonGad.Position(360,0)
HexagonGad.Label("Hexagon",INSIDE)
ComalWindow.InsObject(HexagonGad,Error)
```

To be able to stop the program we have to add a Close gadget. This is also a button gadget and it is placed to the right of all the other gadgets:

```
DIM Close OF ButtonGadget
Close.Size(138,20)
Close.Position(480,0)
Close.Label("STOP",INSIDE)
ComalWindow.InsObject(Close,Error)
```

Now it's time to test for errors:

```
IF Error THEN
  STOP "Could not create one or more gadgets"
ENDIF
```

To prevent the graphics routines from overwriting the gadgets we will change the graphics viewport:

```
viewport(0,width-1,0,height-20)
```

Having made all the gadgets we are entering a loop where we can wait for some buttons to be pressed. The loop terminates when the Close button is pressed:

```
REPEAT
  WAIT
  clearscreen
  IF TriangleGad.Pressed THEN
    triangle
  ELIF SquareGad.Pressed THEN
    square
  ELIF PentagonGad.Pressed THEN
    pentagon
  ELIF HexagonGad.Pressed THEN
    hexagon
  ENDIF
UNTIL Close.Pressed
```

The last statement in the main program closes the graphics system:

```
  textscreen
```

The complete program looks:

```
  // Program 8.1

  USE CITWindow
  USE CITGadgets
  USE Turtle
  USE Figures

  graphicscreen(0)
  ht

  DIM TriangleGad OF ButtonGadget
  TriangleGad.Size(120,20)
  TriangleGad.Position(0,0)
  TriangleGad.Label("Triangle",INSIDE)
  ComalWindow.InsObject(TriangleGad,Error)

  DIM SquareGad OF ButtonGadget
  SquareGad.Size(120,20)
  SquareGad.Position(120,0)
  SquareGad.Label("Square",INSIDE)
  ComalWindow.InsObject(SquareGad,Error)

  DIM PentagonGad OF ButtonGadget
  PentagonGad.Size(120,20)
  PentagonGad.Position(240,0)
  PentagonGad.Label("Pentagon",INSIDE)
  ComalWindow.InsObject(PentagonGad,Error)

  DIM HexagonGad OF ButtonGadget
  HexagonGad.Size(120,20)
  HexagonGad.Position(360,0)
  HexagonGad.Label("Hexagon",INSIDE)
  ComalWindow.InsObject(HexagonGad,Error)

  DIM Close OF ButtonGadget
  Close.Size(138,20)
  Close.Position(480,0)
  Close.Label("STOP",INSIDE)
  ComalWindow.InsObject(Close,Error)

  IF Error THEN
    STOP "Could not create one or more gadgets"
  ENDIF
```

```
viewport(0,width-1,0,height-20)

REPEAT
  WAIT
  clearscreen
  IF TriangleGad.Pressed THEN
    triangle
  ELIF SquareGad.Pressed THEN
    square
  ELIF PentagonGad.Pressed THEN
    pentagon
  ELIF HexagonGad.Pressed THEN
    hexagon
  ENDIF
UNTIL Close.Pressed

textscreen
```

The program is stored on disk under the name Prg8.1. Execute the program to see how it works. Before the program starts you will notice a lot of disk reading. The CIT system is in fact a rather complicated system and a lot of modules are used.

9 Event driven programs.

The program in section 8 can be made better. The final loop is in fact unnecessary. Why not let the buttons themselves call the procedures automatically?

To do this we have to add event procedures to each button. Add the following lines after the the insertion of the TriangleGad (after line 15 in PRG8.1):

```
TriangleGad.EventHandler(TriangleEvent())
PROC TriangleEvent(Id OF USHORT)
  clearscreen
  triangle
ENDPROC TriangleEvent
```

By doing this you are telling CIT that the procedure TriangleEvent should be called automatically when the triangle button is pressed.

Add similar lines after the insertion af the other buttons except the close button.

By using event procedures that are automatically called by CIT the loop can now be changed to this single line:

```
  WHILE NOT Close.Pressed DO WAIT
```

The complete program now looks:

```
  // Program 9.1

  USE CITWindow
  USE CITGadgets
  USE Turtle
  USE Figures

  graphicscreen(0)
  ht

  DIM TriangleGad OF ButtonGadget
  TriangleGad.Size(120,20)
  TriangleGad.Position(0,0)
  TriangleGad.Label("Triangle",INSIDE)
  ComalWindow.InsObject(TriangleGad,Error)
  TriangleGad.EventHandler(TriangleEvent())
  PROC TriangleEvent(Id OF USHORT)
    clearscreen
    triangle
  ENDPROC TriangleEvent

  DIM SquareGad OF ButtonGadget
  SquareGad.Size(120,20)
  SquareGad.Position(120,0)
  SquareGad.Label("Square",INSIDE)
  SquareGad.EventHandler(SquareEvent())
  ComalWindow.InsObject(SquareGad,Error)
  PROC SquareEvent(Id OF USHORT)
    clearscreen
    square
  ENDPROC SquareEvent

  DIM PentagonGad OF ButtonGadget
  PentagonGad.Size(120,20)
  PentagonGad.Position(240,0)
  PentagonGad.Label("Pentagon",INSIDE)
  PentagonGad.EventHandler(PentagonEvent())
  ComalWindow.InsObject(PentagonGad,Error)
  PROC PentagonEvent(Id OF USHORT)
    clearscreen
    pentagon
  ENDPROC PentagonEvent
```

```
  DIM HexagonGad OF ButtonGadget
  HexagonGad.Size(120,20)
  HexagonGad.Position(360,0)
  HexagonGad.Label("Hexagon",INSIDE)
  HexagonGad.EventHandler(HexagonEvent())
  ComalWindow.InsObject(HexagonGad,Error)
  PROC HexagonEvent(Id OF USHORT)
    clearscreen
    hexagon
  ENDPROC HexagonEvent

  DIM Close OF ButtonGadget
  Close.Size(138,20)
  Close.Position(480,0)
  Close.Label("STOP",INSIDE)
  ComalWindow.InsObject(Close,Error)

  IF Error THEN
    STOP "Could not create one or more gadgets"
  ENDIF

  viewport(0,width-1,0,height-21)

  WHILE NOT Close.Pressed DO WAIT

  textscreen
```

Some programmers may find this program rather strange. It looks as if the program does not do anything but waiting for the close gadget to be pressed (in the WHILE statement).

But it does! While you are waiting for the close button to be pressed all your event procedures are called automatically.

This sort of a program is called an event driven program.

The changed version is stored on disk under the name Prg9.1. Execute the program to see if it still does the same.

10 Debugging programs.

The Comal system has an integrated debugger that allow you to examine the behavior of your programs. To show how it works enter this little program (that calculates the number 6!=720):

```
// Program 10.1

n:=6

fak:=1
WHILE n>0 DO
  fak:=fak*n
  n:=n-1
ENDWHILE

PRINT fak
```

or load it from disk (Prg10.1 in the Tutorial drawer).

Select the Trace Mode? menu item in the Program menu (no short form). As a result a little window (the watch window) is opned and the first executable line (here n:=6) is printed in the reverse pen color. The status line shows that you are now in trace mode. In trace mode you can move around in the text but you cannot edit the text and a lot of functions are disabled.

Select the Execute One Step menu item from the Trace menu (short: <right Amiga>+<T>). As a result the current line will be executed and the next line (fak:=1) is printed in the reverse pen color. Continue to execute steps 4-5 times by pressing <right Amiga>+<T>.

As you see, you are able to follow the execution step by step. But to examine the execution in detail it is necessary to watch the value of certain expressions. Select the New Watch Expression menu item from the Trace menu (no short form). A requester will pop up. Write n into the text field of the requester and press <ENTER> to accept. Now the current value of the variable n is shown in the watch window. Select the New Watch Expression menu item once more and enter the name fak. Also the value of this variable will be shown in the watch window as soon as you have accepted it.

Continue to execute program steps (<right Amiga>+<T>) and look at the watch window. The current value of the watch variables are updated after each step.

If you have not stepped to the end of the program (the watch window will close) try to remove a watch expression by double clicking on it in the watch window. You will be asked to confirm.

Step to the end of the program and start tracing again. The actual line is again n:=0. The watch window tells you that the variable is not defined (you has not yet executed the line). Set a break point at line 8 (the line n:=n-1) by double clicking on the line. The line will be marked as a break point line.

Start execution by selecting the Continue Execution menu item from the Program menu (short: <right Amiga>+<G>). The execution starts and it stops the first time the break point line is met. Note the value of the watch expression.

Remove the break point by double clicking on it once more and execute the program to the end.

Let's look at another example. Load the program Prg10.2 from the Tutorial drawer (<right Amiga>+<O>). This program counts the occurences of one (small) text in another (bigger) text.

Enter the trace mode (Program menu) and add the following watches: pos, pattern$, text$ and Cnt. Note that no one is defined yet. Single step (<right Amiga>+<T>) 12 times.

Note that each time the line :

```
  Cnt:=1+Count(pattern$,text$(pos+1..))
```

is executed, the next line to be executed is the first line of the function Count, that is called in this line. This can be avoided by selecting the Execute One Line menu item from the Trace menu (short: <right Amiga>+<L>).

By using the Execute One Line command the whole function and all recursive calls to this function will be executed until execution returns to the same level as before. Try it to see how it works.

III. THE PROGRAMMING ENVIRONMENT

After start Comal will open its own screen and a window in this screen (the editor window) and you may start typing in your program or load a program from disk.

From within the editor you may edit programs, load programs from disk, start executing programs and much more.

1 The keyboard.

The editor works much like any other editor except that it makes syntax control of all the lines entered and organize the program lines with correct indention etc.

All changes will be put into the program buffer immediately. What you see om the screen is the same as is in the buffer. The only exception is the cursor line. It will not be put into the program buffer before you try to leave the line (by using the cursor keys, the ENTER key or the mouse). By pressing <Esc> the line will be restored to the content of the buffer.

Certain keys have special functions. Here is a description of these keys and their functions:

<shift>+<cur up>	move to top of window or show previous page
<Ctrl>+<cur up>	move to start of text
<shift>+<cur down>	move to bottom of window or show next page
<Ctrl>+<cur down>	move to end of text
<Alt>+<cur left>	scroll line right
<Alt>+<cur right>	scroll line left
<Alt>+<cur up>	scroll text down
<Alt>+<cur down>	scroll text up
< <- >	delete character before cursor
<Del>	delete character under cursor
<shift>+<Del>	delete cursor line
<shift>+<Ins>	toggles insert/replace mode - the current state is shown in the status line
<shift>+<Alt>+<Ins>	insert line
<Esc>	restore line

On Amiga's without the numeric keys (Amiga 600) the following replacements for the <Ins> key can be used:

<shift>+<Ins>	->	<shift>+<Alt>+<cur right> or <<Alt>+<RETURN>
<shift>+<Alt>+<Ins>	->	<shift>+<Alt>+<cur down> or <shift>+<Alt>+<RETURN>

2 The mouse.

The mouse can be used to move the cursor around in the the text. If you move to another line the previous cursor line will be syntax checked and if the syntax is not correct an error message will pop up and cursor will not be moved.

On the right side of the editor window you will find a scroll bar. The position of the scroller inside the container shows the approximate position of the window inside the whole text and the size of the scroller shows the size of the window compared to the whole text.

By moving the scroller the window is moved inside the text. Like before this will result in a syntax controll of the previos cursor line and if the syntax is not correct an error message will pop up and the scroller (the window) will not be moved.

3 The menus.

Most of the commands that can be executed from the editor are reached through the menus.

3.1 The Project menu.

The Project menu is placed far to the left.
It contains the following items:

New

By selecting this a new project with its own editor window and its own program buffer will be opened.

Open

A file requester will pop up and you may select the name of a program to load. The program file must be a text file.

After the load the name of the file will be shown in the screen title line.

Save

Store program buffer on disk as a text file. If there is a current name this will be used. Otherwise a requester will pop up.

Save As

Store program buffer on disk as a text file. A requester will pop up so that you can select a name.

File

An extended form of the file requester will pop up. From this it is possible to rename files, delete files, copy files, move files from one directory to another, make a new directory and a new drawer (a direc tory with a drawer info file).

Delete, copy and move accepts the selection of more than one file. Press the shift key while selecting the next file(s).

New Shell

A new Shell (CLI) will open.

Print

Output the program in the program buffer to a printer.

Open Command Window

This will open a command window where you may type in commands like you are used to in other Comal's.
There are no special commands like LIST, ENTER etc. The only commands you can use is PRINT, DIR, CHDIR etc.

About

Useful information about the system will be shown

Clear Program Buffer

The program in the program buffer will be erased. You will be warned if the program in the buffer has not been saved since last change.

Quit Project

The current project will be terminated and if it the only project the Comal system will terminate.

You will be warned if the program in the buffer has not been saved since last change.

3.2 The Edit menu.

The menu to theright of the Project menu is the Edit menu. This menu contains the following items:

Mark Block Start

The cursor line will be one end of a block and it will be printed in the inverse pen color. Select the other end of the block by moving the cursor (cursor up/down, PgUp/Dn) or by moving the scroll bar.

The status line will change from Insert/Replace to Block and you cannot edit the program.

If you are already in block mode this mode will be terminated.

Cut

The marked block will be cut off the text and moved into the clip board.

Copy

A copy of the marked block block will be moved into the clip board

Paste

The content of the clip board will be inserted in the program.

Erase

The marked block will be erased. You will be prompted to confirm.

Insert file
A program text file will be inserted at the cursor line.

Save Block
Store the marked block on disk as a text file.

Print block
Print out the marked block.

Edit Main Program
If the editor contains a module you may turn back to the main program by using this menu.

Comal uses the clipboard.device of the Amiga so that blocks may be moved between different projects (started by the New item in the Project menu) or other text editors using this device.

3.3 The Search menu.

The third menu contains only these three items:

Search
A requester will pop up and you may enter a searh string.

Search & Replace
As before but also a replace string has to be entered.

When a match is found you will be asked if the string should be replaced. After the answer the search will continue. Use Esc to abort.

Search Again
The previous search/search & replace will be resumed.

Note that the search is case sensitive.

3.4 The Macros menu.

A macro is an ARexx script that can be used to add new commands to the editor. The ARexx interface will be described in detail later. Here you will find only a short description of the menu items:

Assign Macro
This item contains a subitem for each function key (F1..F10).
By selecting one of these a requester will pop up an you may enter a macro string to be assigned to that function key. The macro string may be the name of an ARexx script file (which must be placed in the REXX: directory) or an ARexx command.

Load
Load a complete set of assignments for the function key.

Save
Save the current set of assignment for the function keys on disk.

At the start of a project a macro file is read and assignments for the function keys are made according to the definitions in this file. The name of this file is Comal.macro unless another name is specified by the MACRO tool type (se section I.9).

If ARexx is not installed in the system this menu is disabled (goasted).

3.5 The Settings menu.

In the Settings menu various parameters may be changed. The content is:

New Line At <ENTER>

If selected a new line will be inserted after the current line when the enter key is pressed.

Keywords In Capital

Some people prefer small letters. They are more readable they say. With this item you may choose to print keywords in upper or lower case letters.

Create Backup

If selected a backup file will be created when a program is saved.

Create Icon

If selected an icon will be created when a program is saved. By clicking on this icon Comal will be started and the program will be loaded.

All the options selected in this menu wil be stored in the tool type array of the icon.

Store Window Parameters

The window position and size will be stored along with other parameters in the icon (if icon is selected).
This is useful if more than one project is loaded at a time.

ASCII File Format

This item has four subitems:

LF	-	line end is LF (standard Amiga)
CR	-	line end is CR
CR+LF	-	line end is CR+LF
PC	-	program is stored in PC format (line end is CR+LF) and characters are converted.

Automatic variables?

Selects if variables in a program should be created automatically or if they must be declared in a DIM or LOCAL statement before use.

Execute IO Window?

Selects if you want an execute window (used by PRINT, INPUT etc.) to be opened. If this window is not open, Comal will send it the Command Window (if open).

3.6 The Program menu.

Now we come to the importent Program menu:

Control

The program is scanned and the structure is checked. Modules USEd by the program are loaded and initialized and finally all functions/proce dures are made known so that they can be executed from the command window.

Execute

Program execution starts. During execution the status line is changed to Program execution. It is possible to move around in the program buffer during program execution but no editing can be performed.

It is also possible to open another project and start editing or executing a new program.

Stop Execution

Program execution is stopped.

Continue Execution

A stopped program can be restarted.

Parameters { only Workbench 2.04 or higher! }

A special window is opened where you can enter startup parameters send to to the program before start. The form of the parameters are free. The parameters are stored in the Comal structure (see 10.1.2) in the same way as ToolTypes in code files and combined files are stored.

Load

A program stored by the Save item can be loaded.

Save

Store program buffer in code form. This should be used to store modules.

Combine

Combines the interpreter, the program and all modules into an executable file. A file requester will pop up and you may select the name of the output file.

Show Modules

A requester like a file requester shows all modules loaded.
By selecting one of these modules you may see the content.

Remove All Modules

All modules loaded is removed.

Trace Mode

Toggle the trace (debugging) mode.

3.7 The Trace menu.

This last menu is only active if Trace Mode is on.
It contains the following items:

Execute One Step

One programming step is executed. Then all watch expressions in the watch window is updated and control is returned to the editor with the cursor placed on the next line to be executed (shown in reverse pen color).

Execute One Line

One program line is executed. Procedures, functions and single line loops are executed. Then all watch expressions in the watch window is updated and control is returned to the editor with the cursor placed on the next line to be executed (shown in reverse pen color).

Open Watch Window

If you have closed the watch window by clicking on its close gadget it is re-opened.

New Watch Expressing

A requester will pop up and you may enter an expression (like a variable, a function name or a more complicated expression). The expression and its current value will be shown in the watch window and will be updated after each program step.

A watch expression can be removed by double clicking on it in the watch window. You will be prompted to accept.

Clear All Watches

All watch expressions will be removed. You will be prompted to accept.

Clear All Break Points

All break points in the program will be removed. You will be prompted to accept.

A break point is set by double clicking on a program line in the editor while being in trace mode. It is possible to set break points in modules, too.

Concerning the use of the debugger see the tutorial chapter (section 7).

IV. DESCRIPTION OF THE Comal LANGUAGE

1 Data types, variables and expressions.

1.1 Identifiers.

Identifiers are used as names of variables, functions, procedures etc.

Examples of identifiers are:

```
  C   parity    Even    NewName    alfa1    get'key    prg_21
```

An identifier consists of a letter eventually followed by a number of letters, digits, underscores (_) or single quotes ('). An identifier may have any length (in fact not longer than 255 characters, but who would ever reach that limit?). Letters may be national letters such as the danish letters , O and A.

Identifiers may end with a number sign (#) or a dollar sign ($).
Such identifiers are used as names of integer or string variables or functions.

Upper and lower case letters are destinct.
These identifiers are different:

```
  newname     NewName   NEWNAME
```

Some identifiers are reserved and are used either as names of predeclared functions or procedures or they are used as names of Comal language keywords. Examples of reserved words are:

```
  sin    inkey$     IF     PROC     COS     ENDLOOP
```

Reserved identifiers may be written in either upper or lower case letters. This means that both print and PRINT are the name of the output keyword, but Print is not.

1.2 Simple data types.

Comal has a few important data types built into the language. These data types are numbers and strings. Other data types may be defined by the programmer (see the sections 1.3.2 and 1.5).

Numbers may be either integers or floats (reals). But normally you don't need to distinguish between integers and floats.

1.2.1 Constants.

Integers

The following numbers are examples of integers:

```
  123456    -852       0
```

The integer data type is divided into different subtypes differing by the range of integers that can be the value of the type.

These types are:

```
  type      integer range

  ULONG     0 ... 4294967295

  LONG      -2147483648 ... 214783647

  USHORT    0 ... 65535

  SHORT     -32768 ... 32767

  UBYTE     0 ... 255

  BYTE      -128 ... 127
```

Integer constants may be written in either decimal, hexadecimal or binary format. Hexadecimal constants are preceded by a dollar sign ($) and binary constants by a percent sign (%):

```
  $1234FED    $ABCD   $FFFFFFFF
  %010011     %111    %1000111101111110001
```

Floats

The float number type is used to represent fractional numbers and numbers of very large or very small magnitude.

The float numbers are represented internally as floating point (64 bit IEEE). This means that they have aproximately 16 digits of precision and a tens exponent in the range -308 to 308.

Float constants may be written as normal numbers with a single decimal point or in exponential notation:

```
  3.14159     123456.0     -852.3     0.00001
  3.52E-27    4756328E112   0.000123E-27
```

Strings

A string is a sequence af characters (printable as well as non printable).

A string constant is typed as a sequence of characters surrounded by quotes ("):

```
"This is a string containing 51 printable characters"
"Comal"
"123"
```

A quotation mark is written using a double quote:

```
"A ""double quote"" example"
```

Non printable characters are written by using its ASCII number within quotation marks:

```
"A bad error "7""
```

1.2.2 Expressions.

An expression consists of constants, variables, functions, parenthesis and operators that yield a new value.

Expressions involving numbers

Operators used in expressions involving numbers are:

		examples of use
-	unary minus	-7 -3.14157
^	exponentiation	2^3 (=8)
*****	multiplication	5.2*7.1 (=36.92)
/	division	4.8/3 (=1.6)
MOD	modulo (remainder)	27 MOD 4 (=3)
DIV	integer division	27 DIV 4 (=6)
+	addition	4.56+5.77 (=10.33)
-	subtraction	4.56-5.77 (=-1.21)
BITAND	binary AND	%1101 BITAND %0110 (=%0100)
BITOR	binary OR	%1101 BITOR %0110 (=%1111)
BITXOR	binary XOR	%1101 BITXOR %0110 (=%1011)
<	less than	25<37 (=TRUE = 1)
<=	less or equal	27<=27 (=TRUE)
=	equal	5=8 (=FALSE = 0)
>=	greater or equal	-56>=56 (=FALSE)
>	greater than	-33>-57 (=TRUE)
<>	not equal to	44<>0 (=TRUE)
NOT	logical NOT	NOT 44<>0 (=FALSE)
AND	logical AND	44<>0 AND 5=8 (=FALSE)
OR	logical OR	44<>0 AND 5=8 (=TRUE)

The operators are listed in the order of their precedence. This precedence may be changed by using parenthesis:

```
  2+3*4    = 14
(2+3)*4    = 20
```

Before the evaluation of an expression all integer types except ULONGs are changed to LONG. The type of the result depends on the types and the operators in the expression according to the following table

left operand	right operand	result
FLOAT	any number	FLOAT
any number	FLOAT	FLOAT
LONG	LONG or ULONG	LONG
ULONG	LONG or ULONG	ULONG

The result of an expression involving division (/) is FLOAT.

Normally all parts of an expression is evaluated. This means that the right parenthesis in

```
(6-2*3)*(5+7)
```

is evaluated even though the value of the expression is determined by the first parenthesis (which is zero).

An important exception from this rule is expressions containing the logical operators AND and OR. If the left operand of an AND expression is evaluated to FALSE the right operand is not evaluated and if the left operand of an OR expression is evaluated to TRUE, the right operand is not evaluated. As an example the expression

```
(25>37) AND (2=1/0)
```

is evaluated to FALSE and it will not result in a run time error since the right expression is not evaluated.
This evaluation rule for AND and OR simplifies a lot of programming as the following example shows.

Example:

This piece of code can be used to find a given number n in a table of numbers Table()

```
index:=MAXINDEX(Table())
WHILE Index>0 AND n<>Table(index) DO
  index:=index-1
ENDWHILE
```

If the number is not found the value of index is zero.

Note that there are no boolean type in Comal. Boolean expressions evaluates to 1 if they are true and they evaluates to 0 if they are false.

Any number expression used as a boolean expression is considered false if it is zero and true otherwise.

Expressions involving strings

Operators in expressions involving strings are:

		examples of use
(..)	string selection	"strings"(3..6) (= "ring")
*****	string repetition	3*"Comal" (="ComalComalComal")
+	string concatenation	"some"+"times" (="sometimes")
<	less than	"alf"<"alfa" (=TRUE)
<=	less or equal	"abc"<="ABC" (=FALSE)
=	equal	"some"="any" (=FALSE)
>=	greater or equal	"2">="!" (=TRUE)
>	greater than	"Comal">"BASIC" (=TRUE)
<>	not equal to	""<>" " (=TRUE)
IN	contained in	"me" IN "sometimes" (=3)

Note that only the first three has string values. The remaining operators are used between strings but the value of the expression is a number.

The selection operator (..) selects part of a string. The first character of the selected substring is the character with the position specified by the first number in the parenthesis and the last character in the substring is the character with the position specified by the last number.

Example:

The function DATE$ returns the current date in the format yyyy-mm-dd. This piece of code prints the current data in the format mm.dd.yy:

```
PRINT DATE$(6..7),".",DATE$(9..10),"."DATE$(3..4)
```

It is possible to omit the first or the last character position. In this case it is considered as 1 and the length of the string respectively:

```
"sometimes"(..4)    = "some"
"sometimes"(5..)    = "times"
```

During evaluation of expressions involving the comparison oprators <, <=, =, >=, > and <>, the characters in the strings are compared one by one using the ASCII value of the characters (or the value set in the sorting table - see section 10.1.2).

An IN expression evaluates to the first position of the left string operand in the right string operand (if it is part of this) and it evaluates to zero if the left operand is not part of the right operand.

Note that since a non zero value is considered true if it is used as a boolean value, an IN expression can be used as a boolean expression.

Example:

This piece of code can be used to get a "Yes/No"-answer from the user:

```
PRINT "Yes or No (Y/N)? :",
REPEAT
  Answ$:=INKEY$
UNTIL Answ$ IN "YyNn"
PRINT Answer$
```

1.2.3 Variables.

A variable is a location in the main memory used to hold a value of a certain type. Such locations may be identified by a name (an identifier). Very often this name is identified as the variable itself.

Since the variable is used to hold values of a certain type the size of the memory location depends on the type in question.

Number variables

Number variables are declared (memory is reserved) in DIM statements like

```
DIM Number OF FLOAT       // A floating point variable
DIM n OF LONG, m OF LONG  // Two long integer variables
DIM c OF UBYTE            // A variable of type UBYTE
```

The general format of a number declaration DIM statement is

```
DIM variablename [OF NumberType]
```

where NumberType is one of the types FLOAT, ULONG, ... BYTE or a user defined type (see 1.5). The default type is FLOAT.

After the execution of a DIM statements the values of the declared variables are zero. This value may be replaced by another value in assignment statements like:

```
Number:=2.718281828       n:=-38
m:=45+n                   c:=10
```

The general format of an assignment statement is:

```
identifier:=expression
```

The different number types may be mixed in expressions. Comal makes the necessary convertions between the types. If an integer variable is assigned the value of an expression with a floating point value, that value is rounded to the nearest integer value. If a value exceeds the range of the type, an error is reported by the Comal system.

Floating point and long integer variables need not be declared before they are used. If the variable Alfa is not declared before its use, it is automatically created as a floating point variable (unless you have turned off automatic variable creation - see section III.3.6). If the variable i# is not declared before its use, it is automatically created as a long integer variable (the number sign # tells the system that it is a long integer variable).

The value of number variables may be decremented or incremented in special assignment statements like:

```
n:-7
x:+3.2
```

The effect of these statements are the same as the effect af the statements

```
n:=n-7
x:=x+3.2
```

but they are executed a little bit faster.

String variables

String variables are used to hold string values. The name of a string variable always ends with a dollar sign ($). String variables are declared in DIM statements like:

```
DIM Name$ OF 25   // A variable holding up to 25 characters
DIM Answ$er OF 1  // A variable holding only one character
```

In the declaration af a string variable the maximum length of the string value is specified. This maximum string length may be up to 32767. After the declaration the value of the string variable is set to the empty string (""). This value may be replaced by another value in assignment statements like:

```
Name$:="Borge"
Answer$:="y"
```

If a string variable is assigned a string which is longer than the maximum length specified in the declaration statement, the string is truncated.

For example:

```
Answer$:="No" // The string "No" is truncated to "N"
```

A string variable need not be declared before its use. If a string variable is not declared before it is used, the system will automatically reserve space for 80 characters.

The value of string variables may be "incremented" by using a special assignment statement like

```
Name$:+" Christensen"
```

Part of a string variable may be assigned a new value by using the selection operator:

```
s$:="Comal is a programming language"
s$(5..9):=": the"
```

If the value of the new substring is longer than the selected string, the value is truncated. If it is shorter, spaces are added.

1.3 Structured data types.

A structured type is a single type built out of simple types in certain ways and given a single name. This section discuss indexed variables (arrays) and records (strucs). In a later section the class type, which is an extension of the record type, will be discussed.

1.3.1 Indexed variables.

An indexed variable is a sort of table of values all of the same type. The table may have one ore more dimensions.

Indexed variables are declared in DIM statements where the number of indicies (dimensions), the number of index values of each dimension and the type of the elements are specified:

```
DIM a(4,5) OF FLOAT
DIM b(100) OF BYTE
DIM t$(20) OF 30
```

The first table a has two indicies. The first index may have the values 1, 2, 3 and 4 and the second index may have the values 1,2,3,4,5. The total number of elements, which are all of type FLOAT, is 20.

The second table b has only one index taking on values from 1 to 100. The elements are of type BYTE. The third array t$ has one index that can take on values from 1 to 20. Each element is a string with the maximum length 30.

The number of indicies and the range of each index is limited only by the available memory. The specified type may be any type (including user defined types) except pointer type and array type.

The indicies may take on the values from 1 (lower bound) up to the specified upper bound. It is possible to specify other index ranges:

```
DIM a(0..3,0..4) OF FLOAT
DIM b(-50..49) OF BYTE
DIM t$(10..29) OF 30
```

Here is both the upper and lower bound of the indicies specified. The number of index values are the same as in the previous example.

The upper and lower bounds of the index values can be examined by using the functions MAXINDEX and MININDEX:

```
MININDEX(a())     returns 0
MININDEX(a(),1)   returns 0   (same as MININDEX(a())
MAXINDEX(a(),2)   returns 4
MAXINDEX(t$())    returns 29
```

Note that the second argument is optional. If not present the bounds of the first dimensions is returned.

The individual elements of the table is referenced by using the name of the table followed by the indicies of the element in parenthesis:

```
a(2,3):=11.27
INPUT "Enter a number: ":b(35)
t$(15):="blue"
PRINT a(1,1)
```

Each of the elements referenced in this way is treated like a simple variable of the elements type, and they may replace simple variables at any time.

The whole table can be set equal to the same value in one statement:

```
a(,):=111.11
b():=27
t$():="yellow"
```

1.3.2 Strucs (records).

A struc is a user defined type whose value is a collection of values that are (possibly) of different types (in contradiction to indexed variables).

The struc type is defined in a STRUC statement which has the form:

```
STRUC identifier
    :
  DIM statements
    :
ENDSTRUC identifier
```

The DIM statements in the struc definition has the same format as other DIM statements except that all size specifications (like the size of a string and index bounds) must be constants.

Later an extension of the struc concept (classes) will be discussed.

Example:

```
STRUC Person
  DIM FirstName$ OF 20, LastName$ OF 20
  DIM Address$ OF 30, City$ OF 20
  DIM Age of UBYTE
ENDSTRUC Person
```

Note that a STRUC statement defines a new type. No variables are defined. This is done in the normal way using DIM statements.

Example:

```
DIM Person OF Person
DIM Member OF Person
DIM Club(100) OF Person
```

Note that it is possible to use the same name for the variable as is used for the type.

The elements of the struc are called fields and are accessed using the name of the struc variable, a dot and the name of the field (the dot notation).

Example:

```
Person.FirstName$:="Len"
INPUT "Last name: ": Member.LastName$
PRINT Club(35).Age
```

A struc may be assigned the value of another struc in an assignment statement.

Example:

```
Member:=Club(35)
Club():=Person
```

1.4 Dynamic variables. Pointer variables.

Any variable that is to be used in a program needs two things. It needs storage in the main memory to store the value of the variable and it needs some means of identification. A declaration like

```
DIM Alfa OF UBYTE
```

meets both this needs. It creates a variable (reserves storage in main memory) and it identifies this variable by the name Alfa.

There is another more indirect way of creating and identifying a variable. In the declaration

```
DIM p OF POINTER TO UBYTE
```

a variable p is created. The value of this variable is not itself of type UBYTE. In stead the value is the identifier of a variable of type UBYTE.

A variable is in fact a memory location in the main memory. Such a location has an address. This address can be identified by a name (that's what we have used until now) or it can be identified by a number (in our daily life addresses are mixtures of names and numbers). In our newly created variable the address of the UBYTE variable is identified by a number. If you execute the statement

```
PRINT p
```

you will get this number. It turns out to be zero which is synonymous for no address. There is no address since the UBYTE variable has not yet been created.

This can be done in a statement like

```
ALLOCATE(p)
```

Now the statement

```
PRINT p
```

will give a non zero address.

The value of the UBYTE variable is accessed by using the indirection oprator @ as shown in the following statements

```
p@:=5
p@:+7
PRINT p@+27
```

As you see p@ is a normal variable (of type UBYTE).

In the statement

```
ALLOCATE(p)
```

the procedure ALLOCATE reserves space for the variable p@ among all the other variables used in the program. The ALLOCATE procedure may also be called with two parameters:

```
ALLOCATE(p,MemType)
```

If this form is used the memory is allocated in the storage administrated by the Amiga operating system. The value of MemType can be:

```
MEMF_PUBLIC (=1)
MEMF_CHIP   (=2)
MEMF_FAST   (=4)
```

Normally you will use MEMF_PUBLIC. This name as well as the names MEMF_CHIP and MEMF_FAST are defined in the System module.

The memory allocated by the procedure ALLOCATE can be deallocated by the statement

```
DEALLOCATE(p)
```

After the execution of this statement the value of p is again zero (which means no address).

A variable created by the procedure ALLOCATE is created during the execution of the program and is therefore called a dynamic variable. The variable p itself is called a pointer variable since the value of p points to a variable.

The general format of the declaration af a pointer variable is

```
DIM varname OF POINTER TO typename
DIM varname@ OF typename            // Short form
```

where typename can be any type except a pointer type (see next section).

Normally the type pointed at (typename above) has to be known, i.e. the type must be defined somewhere else in the program or it must be imported from a module. Exception from that rule are pointers that are fields in a structure. In this case the type pointed at is set to unknown pointer if it is not defined.

If a structure with unknown pointer types is imported from a module the system looks for the type in the program environment that imports the structure and changes the unknown pointer type to the correct type if it is found. Otherwise the type will contiue to be unknown pointer.

It is not possible to use the indirection operator on a pointer with unknown pointer type.

Example:

To write a text directly into an intuition window a structure of the following is used:

```
STRUC IntuiText
  DIM FrontPen OF UBYTE
  DIM BackPen OF UBYTE
  DIM DrawMode OF UBYTE
  DIM LeftEdge OF SHORT
  DIM TopEdge OF SHORT
  DIM ITextFont OF POINTER TO TextAttr
  DIM IText OF POINTER TO UBYTE
  DIM NextText OF POINTER TO IntuiText
ENDSTRUC IntuiText
```

If the type TextAttr is not defined elsewhere the pointer type is set to unknown pointer.

Example:

In the module System the ComalStruc structure is defined (se section 10.1.2). One of the fields of this structure is:

```
DIM CommPort OF POINTER TO MsgPort
```

The type MsgPort is not known in the module and the type of CommPort is unknown pointer.

If you want to use this field you have to import the module PortObjects along with the module System:

```
USE PortObjects
USE System
```

A pointer variable acts in many cases like a variable of type ULONG. It has a value (an address) of type ULONG, that may be used in any expression. It is also possible to assign a value to a pointer variable using an assignment statement:

```
p:=$0004
```

Such an assignment can be dangerous since you are now able to change any portion of the memory in your Amiga. To avoid mistakes you can only assign an expression of type ULONG to a pointer. In this way most mistakes will be caught by the system because ULONG values are difficult to make (see description of expressions).

Example:

The function in this example implements the PEEK function known from many BASIC dialects

```
FUNC peek(MemPtr OF UbytePtr) OF UBYTE
  RETURN MemPtr@
ENDFUNC peek

TYPE UbytePtr=POINTER TO UBYTE
```

Example:

The following program fraction shows the use of pointers in the creation of a linked list:

{next page}

```
  :
  :

STRUC Person
  DIM Next OF POINTER TO Person
  DIM Name$ OF 30
  DIM Address$ OF 30
  DIM City$ OF 20
  DIM Age of UBYTE
ENDSTRUC Person

DIM Start OF POINTER TO Person
DIM Node OF POINTER TO Person

  :
  :

// Insert the variable Person@ in the list

IF Start=0 THEN
  Start:=Person
ELSE
  Node:=Start
  IF Start@.Name$<Person@.Name$ THEN
    WHILE Node@.Next AND Node@.Next@.Name$<Person@.Name$
      Node:=Node@.Next
    ENDWHILE
  ENDIF
  Person@.Next:=Node@.Next
  Node@.Next:=Person
ENDIF

   :
```

A pointer variable can be used as a one dimensional indexed variable by placing an index value after the variable name. The range of the index values is all non negative integers.

Example:

An array of 100 elements of type Person can be made by executing the statements

```
DIM Pers OF POINTER TO Person

Pers:=malloc(100*SIZE(Person),MEMF_PUBLIC)
```

and you can get one of the elements in this way:

```
Pers(23).Name$="Donald E. Knut"
```

This way of indexing into a dynamic variable can be very useful in connection with some of the system routines.

1.5 Type definition.

New types are defined in a STRUC statement. But new types may also be defined in a TYPE statement. The TYPE statement can take on different forms.

TYPE NewType=OldType

By using this form of the TYPE statement no new type is created. It's only new names for existing types.

Examples:

```
TYPE UWORD=USHORT
TYPE WORD=SHORT
TYPE BOOL=BYTE
```

TYPE PtrType=POINTER TO Type

This form of the TYPE statement is used to create a pointer type.

Examples:

```
TYPE UbytePtr=POINTER TO UBYTE
TYPE WordPtr=POINTER TO WORD
TYPE PersonPtr=POINTER TO Person
TYPE WdPtr=POINTER TO Window
```

TYPE AryType=ARRAY(dimension list) OF Type

This form of the TYPE statement is used to create an array type. The dimension list has the same form as discussed in the description of the DIM statement in section 1.3.1 (Indexed variables) except that only constant may be used.

Examples:

```
TYPE IntArray=ARRAY(0..5,-2..10) OF LONG
TYPE Data=ARRAY(0..150000) OF UBYTE
TYPE String10=ARRAY(0..10) OF UBYTE
```

Having made these type definitions a pointer to (for example) a variable of type Data can be declared and a large array of bytes can be created:

```
DIM DataPtr OF POINTER TO Data
ALLOCATE(DataPtr,MEMF_PUBLIC)
```

```
TYPE FncType=FUNC[(parameter type list)] [OF NumberType]
TYPE PrcType=PROC[(parameter type list)]
```

These forms of the TYPE statement is used to create function and procedure types. This is necessary if you want to make functions or procedures with such parameter types. The terms inside the square brackets are optional.

Examples:

```
TYPE FltFunc=FUNC(FLOAT) OF FLOAT
TYPE ByteFnc=FUNC OF BYTE
TYPE ExceptProc=PROC(ULONG)
```

Example:

In this program an integral function is made and the integral of the function sin from 0 to pi is printed out:

```
PRINT integral(SIN(),0,PI)

FUNC integral(f OF FltFunc,a,b)
  LOCAL n, dx, xi, s
  n:=16; dx:=(b-a)/n;
  xi:=a; s:=(f(a)-f(b))*dx/6
  LOOP n TIMES xi:=xi+dx; s:=s+(f(xi)+2*f(xi-dx/2))*dx/3
  RETURN s
ENDFUNC integral

TYPE FltFunc=FUNC(FLOAT) OF FLOAT
```

TYPE statements may be placed in the main program, in closed procedures/functions or in modules. The program execution speed is not affected by the use of new type names. All type references are resolved at the scanning time.

The search for a matching type is done in this way:

1. The scanner searches for the name in the same program level (the same procedure/function, module or the main program).

2. Then the name is searched for in the modules used in this program level.

3. Then the name is searched for in lower program levels (the main program or a procedure if the current procedure is local) or in modules used in these lower levels.

If the name cannot be found the scanner will stop with an error message and the cursor will be placed on the unknown type name.

2 Program flow control statements.

This section discuss the program statements that can be used to direct the program's sequence of execution ("flow control statements").

2.1 Selection.

Comal provides several forms of decision-making statements: the IF statements and the CASE statement.

2.1.1 The IF statements.

More than five variants of the IF statement are available, depending upon the use of ELSE and ELIF.

Single line IF - THEN

The single line IF statement has the form:

```
IF condition THEN simple_statement
```

where condition is a number expression whose value is interpreted as a boolean value and simple_statement is any non declaration single line statement (except single line IF). If the condition is fulfilled the statement after THEN is executed.

Examples:

```
IF Printer$="Y" THEN SELECT OUTPUT "lp:"

IF x>Max THEN Max:=x
```

IF - THEN - ENDIF

This is the simplest form of the multy line IF statement. The general format is:

```
IF condition THEN
  statements
ENDIF
```

If the condition is fulfilled the statement(s) between the IF-THEN line and the ENDIF line is executed. If the condition is not fulfilled, then these lines are skipped, and the program execution continues after the ENDIF line.

Example:

In the following example an Intuition window is opened. If the operation fails (Window pointer is zero) an error message is printed and the program stops:

```
Window=OpenWindow(ADR(NewWindow))
IF Window=0 THEN
  PRINT "Window could not be opened"
  STOP
ENDIF
```

IF - THEN - ELSE - ENDIF

The general format of this variation of the IF statement is:

```
IF condition THEN
  statements
ELSE
  statements
ENDIF
```

The statement(s) between the IF-THEN line and the ELSE line is executed if the condition is fulfilled, while the statement(s) between the ELSE line and the ENDIF line is executed if the condition is not satisfied. continues after the ENDIF line. One of the statement segments will always be executed, but never both. When the selected segment has been executed the program execution continues after the ENDIF line.

Example:

```
IF Printer$="Y" THEN
  SELECT OUTPUT "lp:"
ELSE
  PAGE  // Clear screen
ENDIF
```

IF - THEN - ELIF - ENDIF

The simplest form of this variation of the IF statement is:

```
IF condition THEN
  statements
ELIF condition THEN
  statements
ENDIF
```

The ELIF (else if) condition is examined if and only if the IF condition is not satisfied. An arbitrary number of ELIF statements may be placed in the IF-statement, for instance:

```
IF condition THEN
  statements
ELIF condition THEN
  statements
ELIF condition THEN
  statements
ELIF condition THEN
  statements
ENDIF
```

The conditions are examined one by one until one is fulfilled or there is no more ELIF lines. If a condition is fulfilled the statement(s) between the corresponding IF/ELIF and the next ELIF/ENDIF is executed. After that the remainig conditions and statements are skipped, and the program execution continues after the ENDIF line. If none of conditions are fulfiled, then no statements are executed, and the program execution continues after the ENDIF line.

Example:

```
IF x>Max THEN
  Max:=x
ELIF x<Min THEN
  Min:=x
ENDIF
```

IF - THEN - ELIF - ELSE - ENDIF

The simplest form of this variation of the IF statement is:

```
IF condition THEN
  statements
ELIF condition THEN
  statements
ELSE
  statements
ENDIF
```

An arbitrary number of ELIF statements may be placed in the IF-statement, for instance:

```
IF condition THEN
  statements
ELIF condition THEN
  statements
ELIF condition THEN
  statements
ELIF condition THEN
  statements
ELSE
  statements
ENDIF
```

The conditions are examined one by one until one is fulfilled or there is no more ELIF lines. If a condition is fulfilled the statement(s) between the corresponding IF/ELIF and the next ELIF/ELSE is executed. After that the remainig part of the IF statement is skipped, and the program execution continues after the ENDIF line. If none of conditions are fulfiled, then the statements between ELSE and ENDIF will be executed.

Example:

This program section prints the solution of a second degree equation (with coefficients A, B and C)

```
D:=B^2-4*A*C    // The discriminant is calculated
IF D<0 THEN
  PRINT "No solution"
ELIF D=0 THEN
  PRINT "There is one solution:";-B/(2*A)
ELSE
  x1:=(-B-SQR(D))/(2*A)
  x2:=(-B+SQR(D))/(2*A)
  PRINT "There are two solutions:";x1;"and";x2
ENDIF
```

2.1.2 The CASE statement.

With a CASE statement it is possible to provide multiple branching. The general form of the statement is:

```
CASE expression OF
WHEN expression_list
  statements
WHEN expression_list
  statements
WHEN expression_list
  statements
WHEN expression_list
  statements
OTHERWISE
  statements
ENDCASE
```

An arbitrary number of WHEN options can be used. The OTHERWISE part is optional.

When the CASE expression has been evaluated, it will be compared with each of the values specified in the first WHEN branch. If one of the values agrees with the CASE expression, then the following statement (until next WHEN/OTHERWISE/ENDCASE) will be executed, and the program execution will continue after ENDCASE.

Each WHEN branch expression in the CASE statement will be examined in the order in which they occur. The first WHEN segment for which an agreement is found will be executed. The remaining part of the CASE statement will be skipped, and the program execution will continue after ENDCASE.

If none of the WHEN branch values agree with the CASE expression, the statement(s) after OTHERWISE (if present) will be executed. If no OTHERWISE part is present and none of the WHEN branch values agree with the CASE expression, and error message will be generated.

Example:

```
CASE month$ OF
WHEN "jan","mar","may","jul","aug","oct","dec"
  days:=31
WHEN "apr","jun","sep","nov"
  days:=30
WHEN "feb"
  days:=28    // No leap year support
ENDCASE
```

The type of the WHEN branch values must be the same as the type of the CASE expression (string or number).

2.2 Repetition.

In Comal there are several forms of repetive statements: REPEAT, WHILE, FOR and LOOP.

REPEAT - UNTIL

This repetive statement allows groups of statements to be executed again and again, until some terminating UNTIL condition is fulfilled.

The general format of the statement is

```
REPEAT
  statements
UNTIL condition
```

Since the the statements are executed before the condition is evaluated, the statements will always be executed at least once.

Example:

```
REPEAT
  CURSOR 5,10
  PRINT ""155"K",   // Erase to end of line
  INPUT "Enter number of disks (between 1 and 10): ": Num
UNTIL Num>=1 AND Num<=10
```

WHILE - ENDWHILE

The WHILE statement is similar to the REPEAT statement. It is used when a block of statements is to be executed repeatedly as long as a particular condition is fulfilled. The major difference is that the condition is evaluated and tested before any statements inside the WHILE-block is executed.

The general format of the WHILE statement is:

```
WHILE condition DO
  statements
ENDWHILE
```

If the condition is fulfilled, then the statement(s) between the WHILE line and ENDWHILE line is executed, and the condition will then be evaluated again.

If the condition is not satisfied, then the statement block is skipped and the program execution will continue after the ENDWHILE line.

Example:

This little program will print the content of a text file on the screen

```
OPEN FILE 1,"TextFile",READ
WHILE NOT EOF(1) DO
  INPUT FILE 1: Line$
  PRINT Line$
ENDWHILE
CLOSE FILE 1
```

Single line WHILE

If only a single statement follows the WHILE line, then the statement can be placed in a single line WHILE.

The format of a single line WHILE is:

```
WHILE condition DO statement
```

where simple_statement is any non declaration single line statement.

Example: This line waits for a key to be pressed without using CPU time:

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

FOR - ENDFOR

The FOR statement is used if a group of statements are to be executed a certain number of times with given counter values. The number of times the statement block is to be executed, is determined by means of a control variable (the counter).

The general format is

```
FOR counter:=start TO end [STEP interval] DO
  statements
ENDFOR counter
```

The FOR statement contains an indication of the start and end values of the control variable. The statement block is executed until the control variable exceeds the the end value.

When the group of statements has been executed, the control variable is changed. How much it is changed can be specified by means of the STEP option. If this option is omitted, then the control variable will be incre mented by 1. After the change of the control variable the value is compared with the end value.

The first time the value is surpassed, the FOR loop is completed , and the program execution continues after the ENDFOR line.

Example:

```
FOR i:=1 TO MaxMember DO
  PRINT Club(i).LastName$,", ",Club(i).FirstName$
  PRINT Club(i).Address$
  PRINT Club(i).City$
  PRINT
ENDFOR i
```

Single line FOR

If only a single statement follows the FOR line, then the statement can be placed in a single line FOR.

The format of a single line FOR is:

```
FOR counter:=start TO end [STEP interval] DO statement
```

where statement is any non declaration single line statement.

Example:

```
FOR x:=0 TO 6.3 STEP 0.1 DO PRINT x;COS(x);SIN(x)
```

LOOP - TIMES - ENDLOOP

The LOOP statement is used if a group of statements are to be executed a certain number of times and you do not need a counter variable.

The general format is:

```
LOOP number TIMES
  statements
ENDLOOP
```

The statements are executed number times before the program execution continues after the ENDLOOP line.

This loop statement is executed considerably faster than the FOR loop because of the lack of the counter variable.

Example:

```
LOOP 4 TIMES
  forward(80)
  right(90)
ENDLOOP
```

Single line LOOP

If only a single statement follows the LOOP-TIMES line, then the statement can be placed in a single line LOOP.

The format of a single line LOOP is:

```
LOOP number TIMES simple_statement
```

where simple_statement is any non declaration single line statement.

Example:

```
t$:=""
LOOP 30 TIMES t$:+"*"
```

Endless LOOP

This is the simplest of all the repetive statements. It is used when a block of statements is to be executed an infinite number of times.

The format of the statement is

```
LOOP
  statements
ENDLOOP
```

Example: The following piece of code is a typical main part of a menu driven program

```
LOOP
  PAGE
  PRINT AT 7,10: "Create"
  PRINT AT 9,10: "Delete"
  PRINT AT 11,10: "Write"
  PRINT AT 15,10: "Select job: ",
  job$:=PressKey$("CcDdWw")
  CASE job$ OF
  WHEN "C","c"
    Create
  WHEN "D","d"
    Delete
  WHEN "W","p"
    Write
  ENDCASE
ENDLOOP
```

The loop can be terminated by executing one of the branching statements (GOTO, IF .. GOTO, EXIT, EXIT WHEN ..).

2.3 Branching.

Branching means changing a program's flow of control by some means other than the statements described in the sections 2.1 and 2.2. There are two such branching statements: the GOTO statement and the EXIT statement.

2.3.1 The GOTO statement.

The GOTO statement is used to redirect execution to another part of the program. The new location is specified by a label.

The general format of a GOTO statement is:

```
GOTO label
```

The format of a label line is:

```
label:
```

where label is an identifier. Note the colon (:) after the label name.

The GOTO statement is not often used, because of the presence of other flexible structures like the other program flow statements discussed in section 2.1 and 2.2. But there are times (usually error conditions) when it is the least painful way to break out of a number of nested program structures.

Example:

```
Window=OpenWindow(ADR(NewWindow))
IF Window=0 THEN
  PRINT "Window could not be opened"
  GOTO CleanUp
ENDIF
```

2.3.1 The EXIT statement.

The EXIT statement is used in any of the repetive statements to cause conditional or unconditional interrupt of the loop. When EXIT has been executed, the program execution continues after the end of the loop.

The format of an EXIT statement is:

```
EXIT [WHEN condition]
```

If the optional WHEN part is present the exit of the loop will only take place if the condition is fulfilled. Otherwise the exit will take place unconditionally.

Example:

```
OPEN 1,"SER:",READ        // Open serial port
LOOP
  ch$=GET$(1,1)
  EXIT WHEN ch$=""26""    // ^Z (ASCII 26) is end of file
  PRINT ch$
ENDLOOP
CLOSE FILE 1
```

3 Procedures and functions.

One way of designing modular programs in Comal is by using procedures and functions.

Procedures and functions are some sort of named subprograms that can be activated in any part of the program.

3.1 Procedures.

The general format of a procedure is

```
PROC procedure_name[(formal_parameter_list)] [CLOSED]
  program_statements
ENDPROC procedure_name
```

The program_statements inside the procedure are activated by an execute statement. The format of this statement is

```
procedure_name[(actual_parameter_list)]
```

The statements inside the procedure are executed as normal program statements. The execution of these statements stops when the ENDPROC statement is reached or when a RETURN statement is executed.

The format of a procedure RETURN statement is

```
RETURN
```

When the execution of the procedure statements stops, the program execution continues after the execute statement that activated the procedure.

The name of a procedure is an identifier. Normally one chooses a name that reflects the action of the procedure. Procedures can be placed averywhere in the program except that it may not appear within IF, CASE, REPEAT, WHILE, FOR, LOOP and TRAP statements.

It's a good idea to place procedures at the end of the program. In this way it reflects the top-down programming strategi.

3.1.1 Procedures without parameters.

The simplest form of procedures are the procedures without parameters. The general format of such procedures is

```
PROC procedure_name
  program_statements
ENDPROC procedure_name
```

and they are activated by an execute statement of the form

```
procedure_name
```

Example:

The procedure EraseToEndOfLine in this example will erase to the end of the line

```
    :
REPEAT
  CURSOR 5,10
  EraseToEndOfLine
  INPUT "Enter number of disks (between 1 and 10): ": Num
UNTIL Num>=1 AND Num<=10
  :

PROC EraseToEndOfLine
  PRINT ""155"K",
ENDPROC EraseToEndOfLine
```

3.1.2 Value parameters.

Data can be transferred to procedures through parameters. There are two kinds of parameters: Value parameters and reference parameters. Value parameters are exclusively used to transfer data into the procedure. Reference parameters are used both to transfere data into the procedure and to get data back from the procedure. Value parameters and reference parameters can be mixed in procedures, but they are discussed in separate sections. Reference parameters will be discussed in the next section.

Value parameters are special kinds of local variables that are initialized when the procedure is called.

A procedure with value parameters have the format

```
PROC procedure_name(formal_parameter_list) [CLOSED]
  program_statements
ENDPROC procedure_name
```

The formal_parameter_list consists of one or more formal parameters separated by commas. The format of a formal value parameter is

```
ParameterName [OF Type]
IntegerName#
StringName$
```

If the type in the first form is not specified, the type of the parameter is set to FLOAT (the default type).

Example:

The procedure polygon in this example draws a polygon. It requires that the Turtle module is loaded.

```
PROC polygon(Corners OF UBYTE, SideLen OF FLOAT)
  LOOP Corners TIMES
    forward(SideLen)
    right(360/Corners)
  ENDLOOP
ENDPROC polygon
```

A procedure with parameter(s) is called by executing a line of the format

```
procedure_name(actual_parameter_list)
```

The actual parameters consists of one or more expressions separated by commas. The types of the expressions must be assignment compatible with the corresponding formal parameters, i.e. numbers must correspond to numbers and strings to strings.

Example:

The following procedure calls are all legal calls to the procedure polygon.

```
DIM s OF BYTE

polygon(3,s)
polygon(4,3.5)
polygon(6,25/3+s)
```

A value parameter can be of any type except a STRUC. An array can be transferred either by defining an array type and then use this type in the formal parameter description, or by using array indicators.

Example:

```
PROC ArrParProc1(Alfa OF ArrType)    // Array type used
PROC ArrParProc2(Beta(,) OF LONG)    // Array indicator used
TYPE ArrType=ARRAY(5,8) OF LONG
```

In both cases the actual parameter must be an array name with array indicators.

Example:

The procedure PrintArray in this example prints an array as a two dimensional table.

```
PROC PrintArray(Beta(,) OF BYTE)
  ZONE 5
  FOR i=1 to MAXINDEX(Beta(,),1) DO
    FOR j=1 TO MAXINDEX(Beta(,),2) DO
      PRINT Beta(i,j),
    ENDFOR j
    PRINT
  ENDFOR i
ENDPROC PrintArray

PrintArray(A(,))
```

Note that if arrays are used as value parameters, a copy of the array is made each time the procedure is called. This can result in a lot of time

consuming copying. Therefor it is recommended to use reference parameters when possible (as for instance in the example above).

Procedures (and functions) can be used as parameters. To do this a procudure (function) type has to be defined (see section 1.5).

3.1.3 Reference parameters.

Reference parameters can be used to return values back from a procedure.

Procedures with reference parameters have the format

```
PROC procedure_name(formal_parameter_list) [CLOSED]
  program_statements
ENDPROC procedure_name
```

The formal_parameter_list consists of one or more formal parameters separated by commas. The format of a formal referencee parameter is

```
REF ParameterName [OF Type]
REF IntegerName#
REF StringName$
```

If the type in the first form is not specified, the type of the parameter is set to FLOAT (the default type).

Example: The procedure swap in this example interchanges the content of two string variables (with maximal length 50)

```
text1$:="Text number 1"
text2$:="Text number 2"
swap(text1$,text2$)

PROC swap(REF t1$,REF t2$)
  LOCAL Temp$ OF 50

  Temp$:=t1$
  t1$:=t2$
  t2$:=Temp$
ENDPROC swap
```

A procedure with reference parameter(s) is called by executing a line of the format

```
procedure_name(actual_parameter_list)
```

The actual parameters consists of one or more variables separated by commas. The type of the variables must be the same as the type of the corresponding formal parameters, i.e. string must correspond to string, FLOAT to FLOAT, ULONG to ULONG etc.

At the call to a procedure with reference parameters a new local variable is not created (as was the case with value parameters). Only a new name is introduced to identify an existing variable. Every change made in the parameter is therefore made on the variable used as the actual parameter in the calling line.

To be specific. The changes made on the strings t1$ and t2$ in the procedure swap in the example above are in fact made on the string variables text1$ and text2$.

3.1.4 Local procedures.

It is possible to place procedures and/or functions inside a procedure (or function). Such a procedure is called a local procedure and it is only known inside the procedure where it is placed.

Example:

```
PROC QuickSort(REF t$(),Start OF LONG,End OF LONG)
  LOCAL temp$ OF 10, x$ OF 10
  LOCAL a OF LONG, z OF LONG

  a:=Start; z:=End; x$:=t$((Start+End)/2)
  REPEAT
    WHILE t$(a)<x$ DO a:+1
    WHILE x$<t$(z) DO z:-1
    IF a<=z THEN
      swap(t$(a),t$(z))
      a:+1; z:-1
    ENDIF
  UNTIL a>z
  IF Start<z THEN QuickSort(t$(),Start,z)
  IF a<End THEN QuickSort(t$(),a,End)

  PROC swap(REF t1$,REF t2$)
    LOCAL Temp$ OF 50

    Temp$:=t1$
    t1$:=t2$
    t2$:=Temp$
  ENDPROC swap

ENDPROC QuickSort
```

3.1.5 Local and global variables. CLOSED procedures.

Variables, procedures, functions and type definitions outside a procedure are accessible from a procedure.

Further more, a variable created inside a procedure (either declared in a DIM statement or declared implicitely in an assignment statement) is in fact created out side the procedure an is still there after the procedure is left.

Example:

The output from the following little program

```
x:=-5
y:=-8
MyProc
PRINT x;y

PROC MyProc
  x:=11
  y:=7
ENDPROC MyProc
```

is

```
11 7
```

Sometimes these effects of creating and/or using variables inside a procedure are intended. But in other cases it is the source of unpredictable and undiserable results.

If a variable used inside a procedure is strictly local to that procedure, it should be declared as local. This is done in a LOCAL statement, that has the same form as a DIM statement:

```
LOCAL name[(dimension_specification)] [OF Type]
LOCAL pntr_name OF POINTER TO Type
LOCAL integer_name#[(dimension_specification)]
LOCAL string_name$[(dimension_specification)] OF string_len
```

Example:

The output from the following little program

```
x:=-5
y:=-8
MyProc
PRINT x;y
```

```
  PROC MyProc
    LOCAL x OF SHORT
    x:=11
    y:=7
  ENDPROC MyProc

is

  -5 7
```

Example:

```
PROC QuickSort(REF t$(),Start OF LONG,End OF LONG)
  LOCAL temp$ OF 10, x$ OF 10
  LOCAL a OF LONG, z OF LONG
  a:=Start; z:=End; x$:=t$((Start+End)/2)
  REPEAT
    WHILE t$(a)<x$ DO a:+1
    WHILE x$<t$(z) DO z:-1
    IF a<=z THEN
      temp$:=t$(a); t$(a):=t$(z); t$(z):=temp$
      a:+1; z:-1
    ENDIF
  UNTIL a>z
  IF Start<z THEN QuickSort(t$(),Start,z)
  IF a<End THEN QuickSort(t$(),a,End)
ENDPROC QuickSort
```

The use of local variables prevents undiserable name coincidence. Another way to avoid this is to make the procedure closed. This is accomplished by adding the word CLOSED at the end of the PROC line.

All variables in a closed procedure are local, and variables, procedures and functions found outside the procedure is unknown inside the procedure.

Example:

The output from the following little program is -5 -8

```
  x:=-5
  y:=-8
  MyProc
  PRINT x;y

  PROC MyProc CLOSED
    x:=11
    y:=7
  ENDPROC MyProc
```

In order to make identifiers outside a closed procedure accessible it is possible to import them. This is done by using the IPMPORT statement or the GLOBAL statement.

The IMPORT statement is used to import identifiers from the program level just below the procedures level (for instance another procedure if the IMPORT statement is placed in a local procedure). The GLOBAL statement imports identifiers from the main part of the current program environment (from the main program, from the initialization part of a module or from the fields in a STRUC if the GLOBAL statement is placed in a method).

The general format of an IMPORT statement and a GLOBAL statement is

```
IMPORT name_list
GLOBAL name_list
```

where name_list is one or more names separated by commas.

Example:

```
IMPORT a,Beta,Str$
GLOBAL p#,Alfa$,x
```

Example:

The QuickSort procedure could have been written (although it is not recommendable):

```
PROC QuickSort(Start OF LONG,End OF LONG) CLOSED
  IMPORT t$
  DIM temp$ OF 10, x$ OF 10
  DIM a OF LONG, z OF LONG
  a:=Start; z:=End; x$:=t$((Start+End)/2)
  REPEAT
    WHILE t$(a)<x$ DO a:+1
    WHILE x$<t$(z) DO z:-1
    IF a<=z THEN
      temp$:=t$(a); t$(a):=t$(z); t$(z):=temp$
      a:+1; z:-1
    ENDIF
  UNTIL a>z
  IF Start<z THEN QuickSort(Start,z)
  IF a<End THEN QuickSort(a,End)
ENDPROC QuickSort
```

It is possible to place USE statements and TYPE definition statements inside a closed procedure.

3.2 Functions.

The FUNC statement is used to define funtions. The general format of a FUNC statement is

```
  FUNC func_name[(formal_parameter_list)] [OF NumType] [CLOSED]
    program_statements
  ENDFUNC func_name
```

or

```
  FUNC int_func_name#[(formal_parameter_list)] [CLOSED]
    program_statements
  ENDFUNC int_func_name#
```

or

```
  FUNC str_func_name$[(formal_parameter_list)] [CLOSED]
    program_statements
  ENDFUNC str_func_name$
```

The default type in the first form is FLOAT. The value returned by a function is calculated in a function RETURN statement of the form:

```
  RETURN expression
```

The expression must match with the function type and there have to be at least one function RETURN line in a function.

The program statements inside the functions are activated by placing the function name in an expression.

Examples:

```
  PRINT 1-errf(1.5)
  y:=sinh(x)/cosh(x)
```

Except for the returned value, there is no difference between functions and procedures. The sections 3.1.1 to 3.1.5 holds for functions, too.

Example:

The following function gcd calculates the greatest common divisor of two integer numbers

```
    FUNC gcd(m OF LONG,n OF LONG)
      IF (m MOD n)=0 THEN
        RETURN n
      ELSE
        RETURN gcd(n,m MOD n)
      ENDIF
    ENDFUNC gcd
```

Example:

The following function WaitKey$ waits for special keys to be entered

```
  :
PRINT "Yes or No (Y/N)? ",
IF WaitKey$("YyNn") IN "Yy" THEN
  :
ELSE
  :
ENDIF
  :

FUNC WaitKey$(WaitChars$)
  LOCAL ch$ OF 1
  REPEAT
    ch$:=INKEY$
  UNTIL ch$ IN WaitChars$
  PRINT ch$,
  RETURN ch$
ENDFUNC WaitKey$
```

Example:

The following function PenColor$ returns a string that can be used to set the drawing color of the console

```
  :
PRINT PenColor$(3),"A nice color",PenColor$(1)
  :

FUNC PenColor$(col OF UBYTE)
  RETURN CHR$($9B)+STR$(30+(col MOD 7))+"m"
ENDFUNC PenColor$
```

4 Exception handling.

This section describes how exceptions such as user break and run time errors may be handled.

4.1 The TRAP statement.

Normally a run time error such as Division by zero or Out of memory will result in a program stop. This can be prevented by the TRAP statement. The format of a TRAP statement is

```
TRAP
  statements_1
HANDLER
  statements_2
ENDTRAP
```

If an error is detected during the execution af statements_1 then the segment statements_2 will be executed. Otherwise the program continues with the first statement after ENDTRAP.

Example:

In this example two files are used. If an error is detected during the opening of the second file the first one has to be closed before the the program stops

```
OPEN FILE 1,File1$,WRITE
TRAP
  OPEN FILE 2,File2$,READ
HANDLER
  CLOSE FILE 1
  REPORT
ENDTRAP

  :
```

In connection with the handling of errors a number of error handling statements an convenient system functions are available:

```
REPORT [ErrNum,[ErrText$]]
```

This statement can be used to generate an error.

If used alone the error reported will be the error that caused the HANDLER section to be executed.

If ErrNum is present the error text connected to this number will be reported.

If ErrText$ is present this test will be reported.

```
RETRY
```

This statement directs the execution back to the first statement in statemens_1.

ERR

A function that returns the error number of the error detected.

ERRFILE

A function that returns the number of the file in question if the error occurs in connection with a file operation.

ERRTEXT$

A function that returns the string containing the error message that the system would have generated.

Example:

This piece of code reads a number using the standard input statement INPUT. If the input is not a legal number the screen is flashed and the input statement is reexecuted:

```
TRAP
  INPUT AT 5,10: "Enter a number: ": Num
HANDLER
  PRINT CHR$(7),
  RETRY
ENDTRAP
```

4.2 The TRAP ESC statement.

If the user presses the Esc-key (while the execute window is active) the program will normally be breaked.

The TRAP ESC statement is used to control the operation of user breaks from within a Comal program.

By executing TRAP ESC+ a user break will be handled in the normal way (the execution is interrupted).

By executing TRAP ESC- the following will occur:

- Pressing the Esc-key will no longer interrupt a running program.
- The system function ESC equals FALSE (=0) until the user tries to break the program; then it equals TRUE (=1).

Thus a Comal program can use the value of ESC to check for user break.

Note, that the selection of the Stop Execution item in the Program menu cannot be masked, i.e. the program will always be stopped by doing this.

5 IO statements.

In this section the different forms of IO statements will be described. The most frequent used IO statements are PRINt and INPUT.

5.1 The PRINT statement.

The general format of the output statement PRINT is

```
PRINT [AT Row,Col:] [USING Format_text:] [print_list] [mark]
```

Simple PRINT statement

The simplest for af the PRINT statement has the form

```
PRINT print_list [mark]
```

where print_list is one ore more of the following separated by either a comma (,) or a semicolon (;)

TAB(position)

where position is a numeric expression with value from 1 to 255

string expression

number expression

and mark is either a comma (,) or a semicolon (;).

The use of TAB will result in the output of spaces up to the specified position if this position is not already passed.

The separator comma (,) has no influence on the output (it acts solely as a separator). The separator semicolon (;) outputs spaces up to the next ZONE tabulation (the default ZONE tabulation is one resulting always in the output one extra space).

If mark is not used a terminating line feed is output. If present it acts like the separator.

Examples:

```
PRINT 2;-3
PRINT "x=",x;"y=",y
PRINT x,TAB(8),
PRINT y,TAB(16)
PRINT SIN(x);
PRINT             // Output an empty line
```

Formatted output with PRINT USING

The PRINT USING statement allows formatted output. This statement has the format:

```
PRINT USING Format_text: [print_list] [mark]
```

where Format_text is a normal text expression containing the format characters '#' '.' and '-' and print_list is one ore more number expressions separated by comma (,).

Examples:

```
PRINT USING "Pi = #.####": PI
FOR x=0 TO 2*PI STEP 0.2 DO
  PRINT USING "sin(#.#)=-#.####":x,SIN(x)
ENDFOR x
```

All the characters in the the format except the special formmat characters are printed as is. The format fields in the text are replaced one by one by the values of the number expressions in the Print_list. To make room for a sign a minus sign (-) has to precede the format characters.

PRINT AT specified position in window

The beginning of the output can be set to any position of the screen by means of the PRINT AT statement with the form

```
PRINT AT Row,Col: [USING Format_text:] [print_list] [mark]
```

Examples:

```
PRINT AT 10,20: "Hello world!"
PRINT AT 0,20: USING "Pi = #.####": PI
```

If Row is zero the current line is used, and if Col is zero the current column is used.

5.2 The INPUT statement.

The general format of the input statement INPUT is

```
INPUT [AT Row,Col[,lenght]:] [Promt-string:] variable_list [mark]
```

Simple INPUT statement

The simple form of the INPUT statement is

```
INPUT [Promt_string:] variable_list [mark]
```

where variable_list is one ore more text or number variables separated by comma (,) and mark has the same effect as in the PRINT statement.

Example:

```
INPUT x,y,t$
INPUT "Enter a text: ": t$
```

If the Promt_string is present this will be printed. Otherwise a question mark (?) wil be printed. Then the cursor is turned on and the user may type in the values for the variables. The input is terminated by pressing the <ENTER> key. The values entered are assigned one by one to the variables in the variable list. A string variable will be assigned the rest of the line entered.

INPUT AT specified position in window

The beginning of the promt string can be set to any position of the window by means of the INPUT AT statement with the form

```
INPUT AT Row,Col: [Promt-string:] variable_list [mark]
```

Example:

```
INPUT AT 5,10: "Enter a number: ": Num
```

If Row is zero the current line is used, and if Col is zero the current column is used (like PRINT AT).

Specifying the length of the input field

By adding a lengthfield to the AT part of an INPUT AT statement the length of the input field can be specified. The general format of this statement is

```
INPUT AT Row,Col,Lenght: [Promt-string:] variable_list [mark]
```

Example:

```
INPUT AT 0,0,1: "Accept (Y/N)? ": Answer$
```

5.3 Redirection of IO.

Normally all IO goes through the console attached to the program. But it is possible to redirect input and/or output to another device or a file. This done by using the SELECT statement that has the form:

```
SELECT Direction File$
```

where Direction is INPUT, OUTPUT or INOUT and File$ is either the name of a file or the name of one of the standard devices

```
ds:   the console window (only output)
kb:   the keyboard (only input)
lp:   a printer (only output)
```

Example:

```
INPUT "Output on printer (Y/N)? ": Answer$

IF "Yy" IN Answer$ THEN
  SELECT OUTPUT "lp:"     // Select printer
  MakeOutput
  SELECT OUTPUT "ds:"     // Output back to window
ELSE
  MakeOutput
ENDIF
```

It is possible to add other devices than the three above (or replace an existing one). To do this you have to write device drivers for that device and then add the device to the device list. Normally these drivers is written in C, but it is possible to write them in Comal, too.
See chapter VII.

5.4 System functions performing IO.

A number of system function can be used to perform IO. These are

FUNC KEY$

By calling this string function the keyboard buffer is scanned. If there is no key in the buffer the empty string is returned. Otherwise the first key in the buffer is returned. Note that some keys return more than a single character.

The key returned (if any) is not echoed on the window.

Example:

```
PRINT "Press any key to continue"
WHILE KEY$="" DO WAIT
```

FUNC INKEY$

This string function does the same as KEY$ but if there is no key in the buffer the empty string is not returned. In stead the process will go to sleep until a key is pressed.

The key returned is not echoed on the window.

FUNC INKEY$(time_out)

This string function does the same as INKEY$ but if there is no key in the buffer it will wait maximal time_out seconds for a keypress before continuing program execution. If no keypress is detected within the time limit the empty strin is returned.

The key returned (if any) is not echoed on the window.

5.5 Other IO related statements and functions.

CURSOR Row,Col

Statement used to place the cursor on the window.

If Row is zero the current line is used, and if Col is zero the current column is used (like PRINT AT and INPUT AT).

Example:

```
CURSOR 15,5
```

CURCOL

Function returning the current column position of the cursor.

Example:

```
CurX:=CURCOL
```

CURROW

Function returning the current row position of the cursor.

Example:

```
CurY:=CURROW
```

PAGE

Statement used to clear the screen or make a form feed (page eject) if the current output direction is the printer.

Example:

```
PAGE
```

ZONE tab_interval

Statement used to set the width of the tab zone on the window (see description of the separator semicolon (;) in section 5.1)

Example:

```
ZONE 8
```

At startup the zone value is 1.

ZONE

Function used to return the current zone value.

Example:

```
OldZone:=ZONE
```

DIGITS number_of_digits

Statement used to specify the number of significant digits wich will be shown in any PRINT statement.

Example:

```
DIGITS 12
```

At startup the number of digits is 10.

DIGITS

Function used to return the current number of digits used in PRINT statements.

Example:

```
OldDigits:=DIGITS
```

6 File statements.

Comal saves data on external storage media using two different types of data files:

- Sequential files
- Random access files (direct files)

Further more data can be saved in two different formats:

- Binary format
- ASCII format (text files)

6.1 Sequential binary files.

In a sequential file one item is saved after another without regard for the space that each item occupies.

A sequential data file has to be opened before you can read from it or write to it. This is done by executing an OPEN statement of the format

```
OPEN FILE FileNo,FileName$,OpenMode
```

where FileNo is a number in the range 1..32767 used to identify the file and OpenMode is one of the following:

```
READ        file is opened for reading only
WRITE       write only (create new file or overwrite existing file)
READWRITE   both read and write access possible
APPEND      write access (append to existing or create a new file)
```

FileName$ is the name of an AmigaDOS file or the name of one of the standard devices introduced in section 5.3. Normally the READWRITE open mode is used only in connection with special devices such as a serial device.

Example:

```
OPEN FILE 2,"MyFile",APPEND
OPEN FILE 1,"Data",READ
```

Data is written to a sequential binary file using a WRITE statement of the form:

```
WRITE FILE FileNo: write_list
```

where write_list is one or more expressions or variables separated by commas (,) and FileNo is the number used in the OPEN statement.

Example:

```
WRITE FILE 1: Alfa,Text$,Beta
WRITE FILE 9: 2+4,3.14159,"Amiga"+"DOS"
```

It is possible to write a whole record or a whole array to a file in one write statement.

Example:

```
DIM Member OF Person
DIM Club(100) OF Person

  :

WRITE FILE 2: Member
WRITE FILE 2: Club()      // Note the array indicator
```

Data is read from a sequential binary file using a READ statement of the form:

```
READ FILE FileNo: read_list
```

where read_list is one or more variables separated by commas (,) and File-No is the number used in the OPEN statement.

Example:

```
READ FILE 1: Alfa,Text$,Beta
READ FILE 9: n,x,t$
READ FILE 2: Member,Club()
```

The type of the variables used to read in data from a file must have the same type as the corresponding data written to the file. It is very important that you observe this rule. Comal has almost no way to test the type of the data saved on a file.

It is recommended that you always use variables (and not expressions) in the WRITE statements. Then you can pair WRITE and READ statements by using the same variables:

```
WRITE FILE 1: Alfa,Text$,Beta

  :

READ FILE 1: Alfa,Text$,Beta
```

When file operations have been completed the file must be closed by using a CLOSE statement of the form:

```
CLOSE [FILE FileNo]
```

If FileNo is specified only this file will be closed. Otherwise all currently opened files will be closed.

Example:

```
CLOSE FILE 2
CLOSE
```

Example:

This little program will print out the whole content of a file containing short integer numbers:

```
DIM n OF SHORT

OPEN FILE 1,"Numbers",READ
WHILE NOT EOF(1) DO
  READ FILE 1: n
  PRINT n
ENDWHILE
CLOSE FILE 1
```

The function EOF used in the program is discussed in section 6.4.

6.2 Random access binary files.

In a random access file data items is stored on fixed positions in the file. A random access file is some times refered to as a direct file, because a specific number of bytes is reserved for each record (a group of data items) allowing direct access to a record when its number is known.

A random access file must exist before you can open it (in contradiction to sequential files opened in WRITE or APPEND mode). A random access file is created by using a CREATE statement:

```
CREATE FileName$,NoOfRecords,RecordLength
```

where FileName$ is the name of the file you are going to create and NoOf-Records is the total number of records each one having the length Record-Length.

The statement creates a file containing NoOfRecords records numbered from 1 to NoOfRecords). The file is filled with CHR$(0).

Example:

```
CREATE "Numbers",100,12   // 100 records each holding one long integer
                          // .. and one float
CREATE "Addresses",50,27  // 50 records each holding a text of max. 25
```

When you are going to create a data file you have to decide how many records you need and to caclulate the necessary size of the records. To make this calculation you have to use this information about how many bytes each data item occupies in the record:

- a text item occupies the actual length of the text (returned by the LEN function) plus two. The last two bytes is used to hold the actual length of the text.

- all other data items occupies the size returned by the function SIZE.

For the simple data types the SIZE function will return:

```
UBYTE and BYTE:   1
USHORT and SHORT: 2
ULONG and LONG:   4
FLOAT:            8
```

Example:

To create a file that is to hold 200 records each holding the content of the structure

```
STRUC Person
  DIM Name$ OF 30
  DIM Address$ OF 30
  DIM City$ OF 20
  DIM Age of UBYTE
ENDSTRUC Person
```

you could do the following:

```
DIM Person of Person
CREATE "PersonFile",200,SIZE(Person)
```

A random access binary file is opened by executing an OPEN statement of the form

```
OPEN FILE FileNo,FileName$,RANDOM RecordLength
```

where FileNo and FileName$ is the same as discussed in connection with sequential files and RecordLength is the size of the records in the file (the same size as was used to create the file).

Example:

```
OPEN FILE 1,"Numbers",RANDOM 4
OPEN FILE 2,"Addresses",RANDOM 27
OPEN FILE 3,"PersonFile",RANDOM SIZE(Person)
```

Data is written to a random access binary file using a WRITE statement of the form:

```
WRITE FILE FileNo,RecordNo[,Offset]: write_list
```

where FileNo and write_list is the same as discussed in connection with sequential files and RecordNo is the number of the record (a positive integer) you want to update.

If Offset is present the writing starts Offset bytes from the start of the record (default is zero). The Offset option is rarely used.

Examples:

```
WRITE FILE 1,51: n,x
WRITE FILE 2,1: Address$
```

Data is read from a random access binary file using a READ statement of the form:

```
READ FILE FileNo,RecordNo[,Offset]: read_list
```

where FileNo and read_list is the same as discussed in connection with sequential files and RecordNo is the number of the record (a positive integer) you want to read.

If Offset is present the reading starts Offset bytes from the start of the record (default is zero). The Offset option is rarely used.

Examples:

```
READ FILE 1,51: n,x
READ FILE 2,1: Address$
```

When file operations have been completed the file must be closed by using a CLOSE statement of the same form as discussed in the preceding section.

6.3 ASCII files.

ASCII files (text files) are always sequential. An ASCII file is opened in the same way as a sequential binary file, e.i. executing a statement of the form:

```
  OPEN FILE FileNo,FileName$,OpenMode
```

where FileNo is a number in the range 1..32767 used to identify the file and OpenMode is one of the following:

```
  READ        file is opened for reading only
  WRITE       write only (create new file or overwrite existing file)
  READWRITE   both read and write access possible
  APPEND      write access (append to existing or create a new file)
```

FileName$ is the name of an AmigaDOS file or the name of one of the standard devices introduced in section 5.3. Normally the READWRITE open mode is used only in connection with special devices such as a serial device.

Example:

```
  OPEN FILE 2,"MyFile",APPEND
  OPEN FILE 1,"Data",READ
  OPEN FILE 7,"SER:",READWRITE    // "SER:" is the AmigaDOS serial device
  OPEN FILE 5,"lp:",WRITE         // "lp:" is the printer device
```

Data is written to an ASCII file using a PRINT statement of the form:

```
  PRINT FILE FileNo: [USING Format_text:] [print_list] [mark]
```

where FileNo is the number used in the OPEN statement. The rest of the statement has the same form as a PRINT statement discussed in section 5.1.

Example:

```
  PRINT FILE 1: Alfa,Text$
  PRINT FILE 5: USING "n = #### x = ##.####": n,x
```

Data is written from an ASCII file using an INPUT statement of the form:

```
  INPUT FILE FileNo: variable_list [mark]
```

where FileNo is the number used in the OPEN statement. The rest of the statement has the same form as a INPUT statement discussed in section 5.2.

Example:

```
INPUT FILE 1: Alfa,Text$
INPUT FILE 5: n,x
```

When file operations have been completed the file must be closed by using a CLOSE statement of the same form as discussed in the preceding sections.

Example:

This little program will print out the whole content of a text file:

```
DIM Line$ OF 150

OPEN FILE 1,"Text",READ
WHILE NOT EOF(1) DO
  INPUT FILE 1: Line$
  PRINT Line$
ENDWHILE
CLOSE FILE 1
```

The function EOF used in the program is discussed in next section.

6.4 File system related functions.

A number of system functions are used in connection with files. These are

```
EOF(FileNo)
```

This function returns TRUE if the internal file pointer of the file opened with file number FileNo is positioned at the end of the file (EOF = End Of File). Otherwise it returns FALSE

EOF returns TRUE as soon as the last data item are read from the file. If an empty file is opened a call to EOF will return TRUE before any reading at all.

The following code fraction shows a safe way to read the whole content of a (possibly empty) file

```
OPEN FILE 1,"Name",READ
WHILE NOT EOF(1) DO
    :
  <do your reading>
    :
ENDWHILE
CLOSE FILE 1
```

```
FREEFILE

  Returns the next available file number. This function is useful in
  library procedures/functions which are to be used in various
  programs.

  Example:

    DIM FileNo OF SHORT

    FileNo:=FREEFILE
    OPEN FILE FileNo,FileName$,APPEND
      :


GET$(FileNo,NumberOfCharaters)

  This string function fetches NumberOfCharaters characters  from the
  file opened with file number FileNo. The length of the string retur-
  ned will be NumberOfCharaters unless the end of file is reached
  first. In this case the string will only be those characters
  retrieved prior to the end of file.
```

Example:

```
  This little program can be used to make a copy of any file
  even a binary file which is not a Comal file)

      DIM Block$ OF 1000

      OPEN FILE 1,Source$,READ
      OPEN FILE 2,Destination$,WRITE
      WHILE NOT EOF(1)
        Block$=GET$(1,1000)
        PRINT FILE 2: Block$,
      ENDWHILE
      CLOSE

    Note the comma at the end of the PRINT FILE statement!
```

6.5 READ and DATA statements.

The READ statement is used to read internal data stored in one or more data statements.

The format of the READ statement is:

```
  READ read_list
```

where read_list is one or more variables separated by commas (,).

When a READ statemet is executed the variables in the READ statement are assigned values from the internal data queue formed by DATA statements in the program.

The format of a DATA statement is:

```
DATA data_list
```

where data_list is one ore more number or string expressions separated by commas (,).

Example:

After the execution of the DATA statement in the following program fraction the variable a will have the value 5, b the value 7, c the value 3 and t$ the value "A string" (without the quotes).

```
READ a,t$,b,c


DATA 5
DATA "A string",7,3
```

It is possible to fill a whole array in one read.

Example:

The following two program fractions are equivalent

```
DIM a(3,4)

FOR i:=1 TO 3 DO
  FOR j:=1 TO 4 DO
    READ a(,)
  ENDFOR j
ENDFOR i

  :

DATA 1,2,3,4
DATA 5,6,7,8
DATA 9,10,11,12
```

and

```
DIM a(3,4)

READ a(,)

  :

DATA 1,2,3,4
DATA 5,6,7,8
DATA 9,10,11,12
```

The values are read one by one in the order the DATA constants are placed in the program. It is possible to change this order by using a RESTORE statement of the form:

```
RESTORE label
```

where label is a label is defined in a label line (see section 2.3.1).

Example:

```
READ a,b
RESTORE d
READ c

DATA 1
d:
DATA 2
```

If you are trying to read beyond the last data constant in the queue the program will stop with an error message. It is possible to test for the end of the data queue by using the function EOD (End Of Data). The function returns true (1) if there is no more data in the queue. Otherwise it returns false (0).

The READ and DATA statements are useful to initialize arrays or structures.

Example:

The NewWindow structure used to open an Intuition window can be initialized in this way:

```
DIM NewWindow OF NewWindow

READ NewWindow

  :

DATA 10,20,400,100,-1,-1  // Potition, size and pen colors
DATA 0                    // No IDCMP flags

DATA WINDOWSIZING BITOR WINDOWDEPTH BITOR WINDOWDRAG
DATA 0,0                  // Gadget and image
DATA ADR("My window")     // Window title
DATA 0,0                  // Screen and bitmap
DATA 100,40,640,200       // Max and min size
DATA WBENCHSCREEN         // Type

  :
```

7 Miscellaneous statements, procedures and functions.

7.1 DOS statements and functions.

A number of statements can be used to execute AmigaDOS commands. These statements are:

CAT

CAT is synonymous for DIR. CAT is relisted as DIR.

CD

CD is synonymous for CHDIR. CD is relisted as CHDIR.

CHDIR

Change the current directory. The format of the statement is:

```
CHDIR path
```

where path is a string expression.

The statement works as the shell command CHDIR.

Example:

```
CHDIR "Demos"
CHDIR "/"
CHDIR Path$+"/Example"
```

COPY

Make a copy of a file. The format of the statement is:

```
COPY OldFile,NewFile
```

where OldFile and NewFile are string expressions.

The statement works as the shell command COPY except that the new name must be specified.

Example:

```
COPY "Demos/Hanoi","Ram:Hanoi"
```

DELETE

Remove a file from disk. The format of the statement is:

```
DELETE FileName
```

where FileName is a string expression.

Example:

```
DELETE "ram:Temp"
```

DIR

Output a directory list of a volume or directory. The format of the statement is

```
DIR path
```

where path is a string expression.

The statement works as the shell command DIR.

Example:

```
DIR                 // The current directory is listed
DIR "Modules"       // The directory Modules is listed
DIR "df0:"+Drawer$
DIR DIR$            // Same as just DIR
```

DIR$

String function returning the complete path of the current directory. The path string will always end with a colon (:) or a slash (/).

Example:

```
PRINT DIR$
Path$:=DIR$
```

MAKEDIR

Synonymous for MKDIR. MAKEDIR is relisted as MKDIR.

MKDIR

Statement used to create a new subdirectory (drawer). The format of the statement is

```
MKDIR name
```

where name is a string expression.

The statement works as the shell command MAKEDIR.

Example:

```
MKDIR "Comal:AsciiFiles"
```

PASS

Statement used to execute shell commands directly. The format is

```
PASS command
```

where command is a string expression. As a result of executing the command a window will open in the Workbench screen and the command will be executed with this window as the output window. After the execution of the command you will be requested to press enter to continue.

If the command string commadn is the empty string a new Shell is opened. Type ENDCLI (or press the close gadget in WB 2.x) to return to your Comal program.

Example:

```
PASS "RELABEL df0: NewName"
PASS ""
```

RENAME

Statement used to give a disk file a new name. The format of the statement is:

```
  COPY OldName,NewName
```

where OldName and NewName are string expressions.

The statement works as the shell command RENAME.

Example:

```
  RENAME "Demos/Hanoi","Demos/Towers"
```

UNIT

Synonymous for CHDIR. UNIT is relisted as CHDIR.

UNIT$

String function returning the name of the volume containing the current directory.

Example:

```
  PRINT UNIT$
  Volume$:=UNIT$
```

7.2 Time related statements and functions.

DATE$

String function returning the current date. The string returned has the format yyyy-mm-dd.

TIME$

String function returning the current time. The string returned has the format hh:mm:ss.

TIMER

Statement used to set the value of an internal constant running timer.

Example:

```
  TIMER 0         // Reset the timer
```

TIMER

Number function used the return the current value of the internal timer.

Example:

```
PRINT TIMER
```

WAIT

Statement used to make your process sleep.
The format of the statement is:

```
WAIT [delaylength]
```

where delaylength is a number expression.

If delaylength is specified the process will sleep for the specified number of seconds. If no delaylength is specified the process will sleep until an event occurs (for instance a key is pressed).

Example:

```
WAIT 5                    // Sleep in 5 seconds
WHILE KEY$="" DO WAIT
```

7.3 Random numbers.

The function RND is used to generate pseudo random numbers. The are two forms of the function RND:

RND

Used without arguments the function RND returns a (pseudo) randaom number in the interval [0;1[(1 not included).

RND(min,max)

Used with arguments an integer random number greater than or equal min and less than or equal max is returned.

The random numbers are read from a random number table (the table values are calculated one by one). To make Comal start a new place in this table

the RANDOMIZE statement may be used. The format of the RANDOMIZE statement is

```
RANDOMIZE [seed]
```

If seed (a numeric expression) is not present the Amiga timer is used to generate the seed.

Example:

The following program will return the same random number sequence each time it is run:

```
RANDOMIZE 100

LOOP 10 TIMES
  PRINT RND
ENDLOOP
```

7.4 STOP and END statements.

A Comal program will stop after the last statement of the program is executed. To make the program stop before the last statement the STOP and END statements may be used. The format of these statements is:

```
STOP [message]
END [message]
```

where message is a string expression whose value is printed in the status line of the editor.

The most importent difference between the STOP and END statements is that the program execution cannot be continued after execution of the END statements.

If the STOP statement is executed the cursor of the editor is placed after the STOP and the message STOP statement is executed is printed in the status line (unless message is present).

Example:

```
STOP "Error while opening the file"
END
```

7.5 Interrupt procedures.

It is possible to make interrupt procedures in Comal. An interrupt procedure is a procedure with the format:

```
PROC Interrupt(SigMask OF ULONG) CLOSED
```

To activate the interrupt procedure a call must be made to the Comal procedure ADDINTERRUPT:

```
ADDINTERRUPT(Interrupt(),SigMask)
```

or

```
ADDINTERRUPT(Interrupt(),SigMask,Priority)
```

where the parameter SigMask is a mask containing the signals that should cause interrupt (more about signals in the Amiga system manual). Interrupt are prioritized. In the second version the parameter Priority the priority sets this priority (-128 .. 127). The default priority used in the first version is 0.

To deactivate the interrupt procedure a call must be made to the Comal procedure REMINTERRUPT:

```
REMINTERRUPT(Interrupt())
```

7.6 Mathematical functions.

ABS absolute value

ABS(x) returns the absolute value (numerical value) of the argument x which may be any real number.

ACS arccosine

ACS(x) returns the arccosine in radians of the argument x which is a real number in the range from -1 to 1.

ASN arcsine

ASN(x) returns the arcsine in radians of the argument x is a real number in the range from -1 to 1.

ATN arctangent

ATN(x) returns the arccosine in radians of the argument x which may be any real number.

COS cosine

COS(x) returns the cosine of the argument x measured in radians.

EXP(x) natural exponential function

EXP(x) returns the number e (=2.718281828..) raised to the power of the argument x.

FLOAT

FLOAT(i) returns a floating point number equalto the argument i which is normally an integer (but may be any number).

INT

INT(x) returns the largest integer integer which is less than or equal to the argument x.

LOG natural logarithm

LOG(x) returns the natural logarithm of the argument x which is a positive real number.

PI

PI is a constant function returning the number pi (=3.1415592655...).

ROUND

ROUND(x) returns the integer which is closest to the argument x.

SGN sign

SGN(x) returns -1 if the argument x is negative, 0 if it is zero and +1 if x is positive.

```
SIN         sine
```

SIN(x) returns the sine of the argument x measured in radians.

```
SQR         square root
```

SQR(x) returns the square root of a non negative argument x.

```
TAN         tangent
```

TAN(x) returns the tangent of the argument x measured in radians.

7.7 Functions involving strings.

```
CHR$
```

CHR$(x) returns a string containing the character with ASCII value x.

```
LEN         string length
```

LEN(t$) returns the length (number of character) of the string expression used as argument.

```
ORD         ordinal value
```

ORD(t$) returns an integer representing the ASCII value of the first character of the string expression used as argument. An error will result if the empty string is used as argument.

```
SPC$
```

SPC$(n) returns the number of blanks (spaces) specified by the argument n.

```
STR$
```

STR$(num) returns the string equivalent of the numeric argument num. The number of significant digits are set by the DIGITS statement.

STR$(format$,num) returns the string equivalent of the numeric argument num. The string argument format$ specifies the format of the string equivalent. See description of PRINT USING in section 5.1 for further details.

Example:

The output from

```
PRINT STR$("#.###",pi)
```

will be

```
3.142
```

VAL

VAL(t$) returns the numeric equivalent of a string representing a number. VAL and STR$ can be concidered as inverse of each other.

7.8 Other functions.

ADR

Function returning the address of the data field of a variable, the code of a procedure/function or the address of the text in a text constant.

Example:

```
ADR(NewWindow)
ADR("My window")
```

FREE

FREE returns the number of free bytes in the work space.

SIZE

SIZE(var) returns the number of bytes occupied by the variable var.

8 Objects.

Very often a computer program can be viewed of as a model of the real world. In these cases a very powerful program design strategi is to base the structures of the program on the structures of the part of the world being modeled. For each object in the real world we should try to construct a corresponding computational object. Such a design strategi is called Object Oriented Programming design strategi (or short OOP).

OOP is still rather unknown and section 8.1 is an introduction to this programming strategi. The sections 8.2-8.4 contains a more formal description of objects.

8.1 Introduction to OOP.

As an example to OOP we are going to make a simple model of a bank account. A bank account is identified by a number (an ID code), there is a balance and an interest. All together these quantities are characterizing a specific bank account and it would be natural to put them into a structure:

```
STRUC BankAccount
  Id OF ULONG
  Balance OF FLOAT
  Interrest OF FLOAT
ENDSTRUC BankAccount
```

Operations can be made on a bank account. In our simple model this is depositing and withdrawal of money which can be implemented by a procedure and a function:

```
PROC Deposit(REF Account OF BankAccount,Amount)
  Account.Balance:+Amount
ENDPROC Deposit

FUNC Withdraw(REF Account OF BankAccount,Amount) OF SHORT
  IF Account.Balance-Amount>=0 THEN
    Account.Balance:-Amount
    RETURN TRUE
  ELSE
    RETURN FALSE
  ENDIF
UNDFUNC Withdraw
```

The function Withdraw returns true if there is enough money on the account to accomodate the withdrawal. Otherwise it returns false.

With these definitions bank accounts can be created by executing DIM statements:

```
DIM MyAccount OF BankAccount
DIM YourAccount OF BankAccount
```

and operations can be made through the procedure Deposit and the function Withdraw:

```
Deposit(MyAccount,1000)

IF Withdraw(YourAccount,1750) THEN
  PRINT "New balance: ",YourAccount.Balance
ELSE
  PRINT "Insufficient founds"
  PRINT "Balance is: ",YourAccount.Balance
ENDIF
```

The operations Deposit and Withdraw are closely bound to the structure BankAccount and it would be natural to put them into the structure definition itself. This is in fact possible and can be done by defining BankAccount like:

```
STRUC BankAccount
  Id OF ULONG
  Balance OF FLOAT
  Interrest OF FLOAT

  PROC Deposit(Amount)
    Balance:+Amount
  ENDPROC Deposit

  FUNC Withdraw(Amount) OF SHORT
    IF Balance-Amount>=0 THEN
      Balance:-Amount
      RETURN TRUE
    ELSE
      RETURN FALSE
    ENDIF
  UNDFUNC Withdraw

ENDSTRUC BankAccount
```

Procedures and functions defined inside a structure are sometimes called methods. Methods cannot be closed but are otherwise normal procedures and functions.

With this new definition bank accounts are still created by executing DIM statements:

```
DIM MyAccount OF BankAccount
DIM YourAccount OF BankAccount
```

and operations can be made through the metods using the dot notation:

```
MyAccount.Deposit(1000)

IF YourAccount.Withdraw(1750) THEN
  PRINT "New balance: ",YourAccount.Balance
ELSE
  PRINT "Insufficient founds"
  PRINT "Balance is: ",YourAccount.Balance
ENDIF
```

By using methods everything that has to do with the bank account is put into the structure definition. This is callded encapsulation and is one of the central elements in OOP. Another important element is the use of the inheritance mechanism to conveniently derive new types from existing types.

A cash credit is a bank account where the balance is allowed to be negative (until some predefined negative value). Apart from this it shares all the proporties of a normal bank account. In making a model of a cash credit it would be natural to try to reuse BankAccount. This may be done in the following way where BankAccount is inherited:

```
STRUC CashCredit
  INHERIT BankAccount

  DIM MinBalance OF FLOAT

  FUNC Withdraw(Amount) OF SHORT
    IF Balance-Amount>=0 THEN
      Balance:-Amount
      RETURN TRUE
    ELSE
      RETURN FALSE
    ENDIF
  UNDFUNC Withdraw

ENDSTRUC CashCredit
```

This new definition of the bank account type can be used like the old one and at the same time:

```
DIM MyAccount OF BankAccount
DIM YourAccount OF BankAccount
DIM Cash OF CashCredit

MyAccount.Deposit(1000)
Cash.Deposit(2000)

IF Cash.Withdraw(1750) THEN
  PRINT "New balance: ",Cash.Balance
ELSE
  PRINT "Insufficient founds or illegal password"
  PRINT "Balance is: ",Cash.Balance
ENDIF
```

Note that the new function Withdraw in CashCredit overloads the original Withdraw function in BankAccount. All other fields or methods of BankAccount are inherited and can be used as if they were defined inside the new type CashCredit.

There is another way to reuse BankAccount in the new CashCredit. First we have to redefine the structure BankAccount:

```
STRUC BankAccount
  Id OF ULONG
  Balance OF FLOAT
  Interrest OF FLOAT

  PROC Deposit(Amount)
    Balance:+Amount
  ENDPROC Deposit

  FUNC Withdraw(Amount) OF SHORT
    Balance:-Amount
    IF Proceed THEN
      RETURN TRUE
    ELSE
      Balance:+Amount
      RETURN FALSE
    ENDIF
  UNDFUNC Withdraw

  FUNC Proceed OF SHORT VIRTUAL
    RETURN Balance>=0
  ENDFUNC Proceed

ENDSTRUC BankAccount
```

The legitimacy of the withdraw transaction is tested by the function Proceed which is declared as VIRTUAL. The meaning of this will be clear soon. Note that allthough BankAccount has been redefined all the code that has used BankAccount until now need not be rewritten.

Now the CashCredit structure can be defined in this way:

```
STRUC CashCredit
  INHERIT BankAccount

  DIM MinBalance OF FLOAT

  FUNC Proceed OF SHORT
    RETURN Balance>=MinBalance
  ENDFUNC Proceed

ENDSTRUC CashCredit
```

To se how this works let us make one normal bank account and one cash credit by executing the statements:

```
DIM Account OF BankAccount
DIM Cash OF CashCredit
```

If the statement

```
IF Cash.Withdraw(750) THEN
```

is executed the method Withdraw of the structure BankAccount is called. Inside this method another method Proceed is activated. But since Proceed in BankAccount is declared as VIRTUAL it is not this one that is called. Instead the method Proceed defined in CashCredit is called and this version of Proceed compares Balance with MinBalance in stead of comparing it with zero.

On the other hand, if the statement

```
IF Account.Withdraw(750) THEN
```

is executed the method Withdraw of the structure Account is again called. But this time the object BankAccount has no descendants and it is the original method Proceed inside BankAccount that is activated.

As a final example let us define a special password-protected bank account where a password has to be supplied to withdraw money. This new password protected bank account shares all the properties of the simple bank account. These properties are inherited from BankAccount in the following definition of ProtectAccount:

```
STRUC ProtectAccount
  INHERIT BankAccount

  DIM Password$ OF 10

  FUNC Proceed OF SHORT
    LOCAL pw$ OF 10

    INPUT "Enter password: ":pw$
    IF pw$=Password$ THEN
      RETURN BankAccount.Proceed
    ELSE
      PRINT "Illegal password"
      RETURN FALSE
    ENDIF
  UNDFUNC Proceed

ENDSTRUC BankAccount
```

8.2 Methods.

An object type is a STRUC type (discussed in section 1.3.2) extended with one or more procedures or functions. The general format of an object type definition is

```
STRUC Identifier

  [USE statements]

  DeclarationStatements

ENDSTRUC Identifier
```

where DeclarationStatements is one or more of the following

- structure DIM statement (see section 1.3.2)
- open procedure
- open function

Procedures and functions in an object type are called methods. Methods must be open but are otherwise normal procedures or functions.

Example:

```
STRUC Person
  DIM FirstName$ OF 20
  DIM LastName$ OF 20
  DIM Address$ OF 30
  DIM City$ OF 20
  DIM Age of UBYTE

  PROC Input
    INPUT "First name: ": FirstName$
    INPUT "Last name:  ": LastName$
    INPUT "Address:    ": Address$
    INPUT "City:       ": City$
    INPUT "Age:        ": Age
  ENDPROC Input

  PROC Print
    PRINT "First name: ", FirstName$
    PRINT "Last name:  ", LastName$
    PRINT "Address:    ", Address$
    PRINT "City:       ", City$
    PRINT "Age:        ", Age
  ENDPROC Print

ENDSTRUC Person
```

A object variable or a pointer to an object must dimensioned in a DIM statement as normal data structures discussed in section 1.3.

Example:

```
DIM Member OF Person
DIM Club(100) OF Person
DIM PersonPtr OF POINTER TO Person

ALLOCATE(PersonPtr)
```

Methods are activated by using the dot notation on the variable.

Example:

```
FOR i:=1 TO 100
  Club(i).Print
ENDFOR i
PersonPtr@.Input
```

The scope of a method is the fields and the other methods inside the actual object.

Example:

The scope of Club(23).Print is the method Input and the fields FrstName$, LastName$, Address$, City$ and Age of Club(23).

An object type is a true extension of the data structure STRUC so that everything that can be done with a data structure can be done with an object type.

8.3 Inheritance.

In the definition of an object type it is possible to inherit fields and methods from a previous defined object types. This is done by placing an INHERIT line in the start of the structure definition.

Example:

In a school you have pupils and teachers. They are all persons but have different additional properties. Pupils are members of a class and have parents, while the teachers teaches in special subjects (like mathematics, computer sience, physics etc.) and they have their monthly salary.

Two object types Pupil and Teacher can be defined in this way:

```
STRUC Pupil
  INHERIT Person

  DIM Class OF BYTE
  DIM Mother$ OF 30
  DIM Father$ OF 30

  PROC Print
    Person.Print
    PRINT
    PRINT "Class:  ",Class
    PRINT "Mother: ",Mother$
    PRINT "Father: ",Father$
  ENDPROC Print

ENDSTRUC Pupil

STRUC Teacher
  INHERIT Person
```

```
    DIM Subject OF UBYTE
    DIM Salary OF FLOAT

    PROC Print
      Person.Print
      PRINT
      PRINT "Subject: ",Subject
      PRINT "Salary:  ",Salary
    ENDPROC Print

  ENDSTRUC Teacher
```

All the fields and all the methods of the inherited are accessible from the descendant object type and from the outside world as if they were directly defined in this object type. If a method in the descendant object has the same name as a method in the ancestor object, the child method will overload the method in the ancestor object. This is the case with the Print method in the example.
The overloaded method may be accessed by using the dot notation.

Example:

A general stack type can be made in this way:

```
  STRUC Stack
    DIM StackTop OF NodePtr

    PROC Push(REF Element OF NodePtr)
      Element@.Next:=StackTop
      StackTop:=Element
    ENDPROC Push

    PROC Pop(REF Element OF NodePtr)
      Element:=StackTop
      IF StackTop<>0 THEN
        StackTop:=StackTop@.Next
    ENDPROC
  ENDSTRUC Stack

  STRUC StackNode
    DIM Next OF POINTER TO StackNode
  ENDSTRUC StackNode

  TYPE NodePtr=POINTER TO StackNode
```

To store a someting on the stack, for example a long integer, we can define a NumberNode:

```
  STRUC NumberNode
    INHERIT StackNode
    DIM Num OF LONG
  ENDSTRUC NumberNode
```

and then execute statements like:

```
  DIM Stack OF Stack
  DIM NumPtr OF POINTER TO NumberNode

  ALLOCATE(NumPtr)
  Stack.Pusk(NumPtr)
```

The elements of the stack need not all be of the same type. They only need to be objects that are descendants of the StackNode object.

A structure A is assignment compatible to a structure B, i.e. an assignment of the form A:=B can be made, if A and B have the same type or if A's type is the ancestor of B's type. That's why the variable NumPtr in the example can be used as the actual parameter to the formal parameter Element in the method Push.

8.4 Virtual methods.

If the Print method was called from the Input method in the Pupil object in the section 9.3, it would have been the ancestors Print method, i.e. the fields of the descendant (Class, Mother$ and Father$) would not be printed.

It is possible to force the system to call the method of the youngest descendant (if any) by specifying the method as virtual. A virtual method is defined by using a procedure/function head of the form:

```
  PROC func_name[(formal_parameter_list)] VIRTUAL
```

or

```
  FUNC func_name[(formal_parameter_list)] [OF NumType] VIRTUAL
```

Example:

This example shows a definition of different figures constituing a class hierarchy. The point is the base figure. The circle and the ractangle shares properties with this but each one has its own characteristics:

```
  STRUC Point
    USE Graphics

    DIM PosX OF FLOAT, PosY OF FLOAT
    DIM FigureColor OF BYTE
```

```
  PROC Init(x,y,Color OF BYTE)
    PosX:=x; PosY:=y
    FigureColor:=Color
    pencolor(FigureColor)
    moveto(x,y)
    Draw
  ENDPROC Init

  PROC Move(dx,dy)
    pencolor(0)
    moveto(PosX,PosY)
    Draw
    pencolor(FigureColor)
    PosX:+dx; PosY:+dy
    moveto(PosX,PosY)
    Draw
  ENDPROC Move

  PROC Draw VIRTUAL   // Virtual method
    plot(PosX,PosY)   // Defines the form of the figure
  ENDPROC Draw

ENDSTRUC Point

STRUC Circle
  INHERIT Point

  USE Graphics

  DIM Radius OF FLOAT

  PROC Init(x,y,Color OF BYTE,R)
    Radius:=R
    Point.Init(x,y,Color)
  ENDPROC Init

  PROC Draw
    circle(PosX,PosY,Radius)
  ENDPROC Draw
ENDSTRUC Circle

STRUC Rectangle
  INHERIT Point

  USE Graphics

  DIM Height OF FLOAT, Width OF FLOAT

  PROC Init(x,y,Color OF BYTE,h,w)
    Height:=h
```

```
    Width:=w
    Point.Init(x,y,Color)
  ENDPROC Init

  PROC Draw
    draw(Width,0)
    draw(0,Height)
    draw(-Width,0)
    draw(0,-Height)
  ENDPROC Draw

ENDSTRUC Rectangle


STRUC Square
  INHERIT Rectangle

  PROC Init(x,y,Color OF BYTE,s)
    Rectangle.Init(x,y,Color,s,s)
  ENDPROC Init

ENDSTRUC Square
```

If you are using pointers to structures containing virtual methods it may be necessary to store a method table in the data field of the structure. You have to tell Comal that you want this table by using the line

```
METHODTABLE
```

in the start of the structure.

Example:

This structure is the ScreenClass defined in the module CITScreen:

```
STRUC ScreenClass
  METHODTABLE

  FUNC CreateObject(REF Screen OF Screen) OF SHORT VIRTUAL
    RETURN TRUE
  ENDFUNC CreateObject

  PROC DeleteObject(REF Screen OF Screen) VIRTUAL
  ENDPROC DeleteObject

ENDSTRUC ScreenClass
```

NOTE: If a method table is stored in the data field you cannot use the structure in a call to an external (C programmed) routine (for instance a system routine).

8.5 Constructors.

Sometimes an object needs to be initialized before it can be used.

Example:

One of the basic structures of the Amiga operating system is the List structure:

```
STRUC List
  DIM lh_Head OF POINTER TO Node
  DIM lh_Tail OF POINTER TO Node
  DIM lh_TailPred OF POINTER TO Node
  DIM lh_Type OF UBYTE
  DIM lh_pad OF UBYTE
ENDSTRUC List
```

which is a double ended list used to store nodes of the form:

```
STRUC Node
  DIM ln_Succ OF POINTER TO Node
  DIM ln_Pred OF POINTER TO Node
  DIM ln_Type OF BYTE
  DIM ln_Pri OF BYTE
  DIM ln_Name OF ULONG
ENDSTRUC Node
```

Before the list can be used it has to be initialized to an empty list by executing statements like:

```
lh_Head:=ADR(lh_Tail)
lh_TailPred:=ADR(lh_Head)
lh_Tail:=0
```

If the initiatization of an object does not require parameters (like the initialization of List in the example) it can be put into a method that is declared as a constructor method. A constructor method is a method with a head of the form:

```
FUNC func_name CONSTRUCTOR
```

Such a method is called automatically when the object is created as a static variable in a DIM or LOCAL statement or is created as a dynamic variable by executing the ALLOCATE procedure.

A constructor returns a SHORT integer value to be considered as a boolean value. TRUE (=1) if the initialization succeded. FALSE (=0) otherwise.

Example:

A smarter definition of the List object is the following:

```
STRUC List
  DIM lh_Head OF POINTER TO Node
  DIM lh_Tail OF POINTER TO Node
  DIM lh_TailPred OF POINTER TO Node
  DIM lh_Type OF UBYTE
  DIM lh_pad OF UBYTE

  FUNC New CONSTRUCTOR
    lh_Head:=ADR(lh_Tail)
    lh_TailPred:=ADR(lh_Head)
    lh_Tail:=0
    RETURN TRUE
  ENDFUNC New

ENDSTRUC List
```

The List in the module ExecLists found in the directory SystemModules is defined in this way.

If a field is an object containing a constructor this constructor is called, too. If an object inherits other objects all the ancestors constructor methods are called when an object of that type is created. But you should note that the virtual method calling mechanism does not function at the initialization time (the descendants are not initialized until after all the ancestors have been initialized).

If a constructor returns FALSE, all the destructors (see next section) of the sub objects (fields or inherited objects) that has been successfully initialized, will be called.

8.6 Destructors.

If an object needs to do some cleaning up before its data fields are removed you can declare a method as a destructor. A destructor method is a method of the form:

```
PROC proc_name DESTRUCTOR
```

Such a method is called whenever the memory used by the objects data field is removed. This will be the case if DEALLOCATE is called or at a

return from a procedure or function (if the object is a local variable). But a constructor may also be called when all the variables are cleared, i.e after the program has stopped!

Because destructors are called in many different sitiuations a destructor method should be as short as possible, it should do no IO and it should be free of errors!

The best way to develop constructors is to first activate them manually (leave out the word CONSTRUCTOR). Then, when you think it works, add the word CONSTRUCTOR.

Example:

One of the basic elements of the Amiga multy tasking system is the message port. A message port consists of a list node (such that it can be put into the list of public ports if desired) and some fields.

One of the fields contains the number of a signal bit that has to be allocated when the port is created and it has to be deallocated when the port is removed. Another field is a list where messages can be placed. Also this has to be initialized.

The alocation of a signal bit can be done in a constructor and the deal-location in a destructor. A message port can be defined in this way:

```
STRUC MsgPort
  INHERIT Node

  USE ExecLibrary
  USE ExecLists

  DIM mp_Flags OF UBYTE
  DIM mp_SigBit OF BYTE
  DIM mp_SigTask OF ULONG
  DIM mp_MsgList OF List

  FUNC Init CONSTRUCTOR
    ln_Type:=NT_MSGPORT
    mp_SigTask:=FindTask($0)
    mp_SigBit:=AllocSignal(-1)
    RETURN mp_SigBit>=0         // No signals if mp_SigBit<0
  ENDFUNC Init

  PROC Delete DESTRUCTOR
    IF ln_Name<>0 THEN
      RemPort(ADR(Node))
      ln_Name:=0
    ENDIF
    IF mp_SigBit>=0 THEN
      FreeSignal(mp_SigBit)
    ENDIF
  ENDPROC Delete
ENDSTRUC MsgPort
```

Note that the list mp_MsgList is not initialized in the constructor shown. But this list is imported from the module ExecLists and the list in this moule contains a constructor that will do the initialization.

9 Modules.

A module is a program structure used to devide a program into smaller parts. A module contains variables, procedures, functions and type definitions that can be use outside the module.

9.1 Making modules.

A module has the form:

```
MODULE ModuleName

  [USE statements]

  [EXPORT statements]

  <normal program lines>

ENDMODULE ModuleName
```

Example:

```
MODULE Hyperbolic

  EXPORT sinh,cosh

  FUNC sinh(x)
    RETURN (EXP(x)-EXP(-x))/2
  ENDFUNC sinh

  FUNC cosh(x)
    RETURN (EXP(x)+EXP(-x))/2
  ENDFUNC cosh

ENDMODULE Hyperbolic
```

All procedures, functions and variables are strictly local to the module. Only those names that are exported in EXPORT statements can be used outside the module. An EXPORT statement has the form:

```
EXPORT ExportList
```

where ExportList is one or more of the following separated by commas (,):

```
Identifier [AS ExportName]
```

If the AS part is present the ExportName is used in references to the exported element. Otherwise it is referenced by the name used inside the module itself.

Example:

If the EXPORT statement in the previous example is changed to

```
EXPORT sinh,sinh AS SINH,cosh,cosh AS COSH
```

both sinh and SINH can be used to refer to the sinh function (and similar for cosh).

It is possible to export names that are imported from other modules. This is called re-export.

A module can be placed in the program that uses the module (local module) or it can be placed in a separate disk file (external modules). There is no limit to the number of modules that can be placed in a program or in a disk file.

An external module is best made by first make it as a local module. Then, when you think it works, store it on disk in code form (open a new project, use the clipboard to move the module to the new project and then save the module using the Save item in the Program menu).

9.2 Using modules.

The exported names in a module (including type names) are made accessible by a USE statement. A USE statement has form:

```
USE ModuleName [FROM FileName]
```

Example:

```
USE System
USE Hyperbolic FROM MathModule
```

All USE statements are executed at the scanning time. If the USE statement is used without the FROM part the system searches for the module in the following way:

1. among the local modules
2. in a file Comal:Modules/ModuleName.mod (if it exists)
3. in a file Comal:SystemModules/ModuleName.mod (if it exists)
4. in a file Comal:Modules/ModuleName (mashine coded module)
5. in a file Comal:SystemModules/ModuleName (mashine coded module)

If the file in 2. or 3. is found there must be one module by name ModuleName in this file.

If the second form of the USE statement is used, the specified file is loaded and if it is a Comal module (.mod file) there must be one module by name ModuleName in this file.

A USE statement can be placed in a program, in an object, in a closed procedure or function or in a module, and all the type names and all the exported names in the module used can be accessed in the scope of the USE statement.

9.3 Signal procedures.

It is possible to make the module recieve messages about changes in the system. These messages are send as signal numbers to a procedure declared as a signal procedure.

The head of a signal procedure has the form:

```
PROC ProcName(Sig OF LONG) SIGNAL
```

Example:

```
PROC Signal(s OF LONG) SIGNAL
  CASE s OF
  WHEN SIG_DISCARD
    CleanUp
  OTHERWISE
    // No action
  ENDCASE
ENDPROC Signal
```

The signal numbers send and their meanings are:

```
    SIG_CLOSE (1)        send if interpreter is going to close
    SIG_DISCARD (2)      send if the module is being removed
    SIG_CLEAR (3)        send if all variables in main program are cleared
    SIG_RUN (4)          send before execution starts
    SIG_STOP (5)         send at program stop that can be continued
    SIG_END (6)          send at program stop that cannot be continued
    SIG_STARTTRACE (7)   send at change to trace mode
    SIG_STOPTRACE (8)    send if trace mode ends
    SIG_STEP (9)         send if a step is executed in trace mode
    SIG_CONTINUE (10)    send if program execution continues after stop
```

The symbolic names used are defined in the System module.

NOTE! Signal numbers send to a module should not be confused with signal numbers used in the Amiga operating system.

10 Description of the standard modules.

In the directory SystemModules a number of useful modules are found. The content of some of these modules will be described in this section.

10.1 The System modules.

As is the case with other implementations of Comal the system module contains implementation dependent stuff. In this implementation the system module is devided into two parts: A module written in Comal by name System, and a machine coded module by name SystemCode.

10.1.1 SystemCode.

This module contains the following procedures and functions:

```
FUNC ComalStrucAdr OF ULONG
```

Returns the address of the ComalStruc (see description of the System module in next section).

```
FUNC CharArrayToString$(Array OF ULONG)
```

Converts a null terminated array of characters into a Comal string.

```
FUNC ComalWait(SignalMask OF ULONG) OF ULONG
```

This function is used like the function Wait in the Exec library of the Amiga operating system.

The function OR's all the Comal signals to the actual parameter Signal-Mask and then calls Wait. Among other things this means that it is possible to break a Comal program that has called ComalWait (this cannot be done if Wait is called!)

The return value is a mask containing the signals that caused the return (and this may be one of the Comal signals).

```
FUNC SysLibVersion OF USHORT
```

Returns the current version of your system.

```
FUNC LockComalWindow OF ULONG
```

Get a shared lock for the standard IO window. The return value is a pointer to the standard IO window.

```
PROC UnlockComalWindow
```

Call this procedure when the pointer returned by LockComalWindow is not needed any longer.

```
PROC AddSignal(SigMask OF ULONG)
```

Call this routine if you want one or more signals to be a Comal signal. The arrival of a Comal signal will cause ComalWait to return and the sleep state started by the WAIT statement (see sectio 7.2) will be terminated. The actual parameter is a mask of signals.

```
PROC RemSignal(SigMask OF ULONG)
```

Removes a Comal signal added by AddSignal.

```
PROC AddComalDevice(IoDevice OF ULONG)
```

Call this procedure if you want to add a new IO device to the list of standard devices (like "ds:", "kb:" and "lp:"). The actual parameter must be a pointer to an initialized device structure.

```
PROC RemComalDevice(IoDevice OF ULONG)

  Call this with a pointer to the IO device you want to remove from the
  list of IO devices.


PROC AddExcept(ExceptStruc OF ULONG)

  Call this procedure if you want to add an exception routine to the
  list of exception routines. The actual parameter must be  a pointer
  to an initialized exception structure.


PROC RemExcept(ExceptStruc OF ULONG)

  Call this with a pointer to the exception structure you want to remove
  from the list of exception routines.
```

Although it is possible to use a Comal procedure as an exception routine the last two routines are normally not used from Comal. Use the procedures ADDINTERRUPT and REMINTERRUPT in stead (see section 7.5).

10.1.2 System.

The System module uses the SystemCode module from where the functions CharArrayToString$ and ComalWait is re-exported.
Two routines are exported:

```
FUNC ComalPath$

  Returns a string containing the complete path of your comal system.


FUNC StartupArg$(n OF USHORT)

  Returns the n'the startup argument (if started from the Shell) or the
  n'the tool type (if started from Workbench). If n is out of range
  (zero or too large) the empty string is returned.

  For the time being this function is of no interest if the Comal
  program is started from the editor.
```

In this module the Comal structure is defined:

```
STRUC ComalStruc
  DIM BreakFlags OF UBYTE
  DIM SecBreakFlags OF UBYTE
  DIM Flags OF USHORT
  DIM CurrWorkBottom OF POINTER TO UBYTE
  DIM CurrWorkTop OF POINTER TO UBYTE
  DIM MinStack OF POINTER TO UBYTE
  DIM IO_Screen OF POINTER TO Screen
  DIM CommPort OF POINTER TO MsgPort
  DIM AsciiId OF POINTER TO UBYTE
  DIM MainPrgBuf OF POINTER TO PrgBuf
  DIM PrgEnv OF POINTER TO PrgEnv
  DIM WorkStart OF ULONG
  DIM WorkEnd OF ULONG
  DIM WorkLength OF ULONG
  DIM SortTable OF POINTER TO UBYTE
  DIM ComalPath OF POINTER TO UBYTE
  DIM Parameters OF POINTER TO ULONG
ENDSTRUC ComalStruc
```

Only a few fields are of interest for Comal programmers:

IO_Screen

A pointer to the screen where all windows should be placed. This pointer is non zero even if the screen is the Workbench screen.

CommPort

If non zero this is a pointer to a message port allocated for you.

AsciiId

This is a pointer to a null terminated array of characters that identifies this interpreter process.

SortTable

This is a pointer to a table of sort values used when strings are compared. Normally it is filled with the ASCII values.

It is possible to change the content of the table (do not change the pointer itself!) with other sorting values.

Example:

By using this little code upper and lower case letters will have the same sorting values:

```
FOR i:=ORD("a") TO ORD("z") DO
  ComalStruc@.SortTable(i):=i-32
ENDFOR i
```

ComalPath

This is a pointer to a null terminated string containing the complete path of your comal system.

Parameters

This is a pointer to an array of pointers to null terninated strings (the startup arguments). The array is terminated by a null pointer.

A pointer ComalStruc to the Comal structure is exported.

The following useful types are defined in the module:

```
TYPE BOOL=BYTE
TYPE bool=BYTE
TYPE WORD=SHORT
TYPE word=SHORT
TYPE UWORD=USHORT
TYPE uword=USHORT
```

The signal numbers SIG_CLOSE, SIG_DISCARD etc. (see section 9.3) and the memory types MEMF_PUBLIC etc. (see section 1.4) are defined and exported from the module.

10.2 The graphics modules.

Two graphics modules Graphics and Turtle can be used.

10.2.1 Graphics.

This module is compatible with graphics modules (packages) found in other implementations of Comal (UniComal for the C64 and PC'c and Amiga-COMAL v. 2.x).

The Graphics module 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 blue (red in WB 1.3)
```

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 unless 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 the procedure 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 (se 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 coordinates (x0+x,y0+y).

The new pen position is the point with 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 coordinates (x,y).

The new pen position is the point with coordinates (x,y).

```
PROC fill(x,y)
```

Fills an area with the current pencolor. The area to be filled must contain the point (x,y) and be bounded by either a curve in a color different from the background color or by the frame set by the clip.

The current pen position is unchanged.

Example:

```
pencolor(1)
circle(300,100,50)  // Draw a circle with pencolor 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(Window)
```

Opens the graphics system. If the actual parameter is zero the standard IO-window will be used as the graphics window. If the parameter is non zero it must be the address of a window and this window will be used as the graphics window.

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 brought 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	0/1	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 ILBM-picture previous stored by Comal (save-screen and SaveWindow procedures) or by another program (like DeLuxe Paint) to be fetched from disk and placed in the graphics window.

Example:

```
loadscreen("MandelBrot.ilbm")
```

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 edges in pencolor 3
```

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 black (white in WB 1.3) line
pencolor(3)
circle(300,100,50)  // .. a red circle
paint(300,100)      // .. and fill the circle with blue 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.

Note that if the graphics window is placed in the IO-window (this is the case if the procedure graphicscreen is called with a zero as parameter) all the Comal windows will change colors. The proceudre is implemented to insure compatibility with UniComals graphics module. 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,"Comal")
```

```
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 transparent polygon will be drawn).

Example:

```
DIM x(3), y(3)
READ x(),y()
interiorcolor(3)
outlinecolor(2)
polygon(3,x(),y()) // Draw a triangle in col 3 with edges in color 2
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 window to the printer.

PROC savescreen(name$)

The procedure causes the content of the graphics window to be saved on disk in ILBM format.

Example:

```
savescreen("MandelBrot.ilbm")
```

PROC SaveWindow(name$,mode)

The procedure works as savescreen above if the parameter mode has the value 0. If mode has the value 1 the window is saved in compressed form (using the ByteRun1 compressing algorithm). This may reduce the size significantly.

Example:

```
SaveWindow("Mandelbrot.ilbm",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)
```

color number 1 (set by pencolor procedure) has the color magenta.

The colors of all the color registers may be restore to their original value by palette(-1).

PROC TextAttr(attr)

The procedure defines the manner in which the printout from the procedure TextAttr will appear on the graphics window.

The following values may be used for the parameter attr:

```
  0  normal text
  1  bold
  2  italic
  4  underlined
  8  inverse
 16  only text (no background)
```

More values can be set by adding the values.

Example:

```
  TextAttr(1+2) // Print in bold and italic
```

```
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(width#,height#,direction#,mode#)
```

The proceudre is implemented to insure compatibility with UniComals graphics module and only the last parameter is significant. The meaning of this is:

```
  0  print without background
  1  print with background (as attr=16 in TextAttr 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 co-ordinates) 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 coordi-
  nates (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.
```

10.2.2 Turtle.

```
This module extends the Graphics module with relative graphics routines
(Turtle graphics). The module is written in Comal and uses all the
routines in the Graphics module.

The command graphicscreen executes an implicit window(-160,160,-100,100)
and draws a turtle (a small triangle) in the mittle of the window.

All procedures and functions in the Graphics module are accessible from
the Turtle module 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 parame-
    ter 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 direction is changed.

```
PROC back(x)
```

The procedure moves the pen x units backwards in 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 gra-
  phics system. The parameter no must be an integral number in the ran-
  ge 0-33. The meaning of the returned value can be found in the follo-
  wing table:

  number    information about     range     Affected by
    0       graphics mode         0/1       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/showturtle
   12       pen inside dr. frame  0/1       most drawing procs
   13                             1
   14                             0
   15       wrapping active       0         (wrap not implemented)
   16       pen down              0/1       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                             1
   28                             0
   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 as 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 module 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 the Graphics module will run without changes with the Turtle module. 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)
```

10.3 Catalog.

This module treats a directory mush like a file. The module exports the following procedures and functions:

```
PROC OpenCatalog(Name$, REF CatId OF ULONG)
```

Opens a volume or directory for subsequent readings. The returned value in CatId is used as an identifier for the catalog.

```
PROC ReadCatalog(CatId OF ULONG,REF name$,REF filesize OF LONG)
```

Read the next file/subdirectory from the catalog. If filesize=-1 this is a subdirectory otherwise it is the size of the file.

```
PROC CloseCatalog(number OF ULONG)
```

Close a previosly opened catalog.

```
FUNC EOC(number OF ULONG) OF BOOL
```

Returns TRUE (=1) if no more directories. Otherwise FALSE (=0).

```
PROC CloseAllCatalogs

  Close all catalogs opened by OpenCatalog.
```

10.4 Bob and sprite modules.

The three modules SpriteRoutines, SpriteStrucs and GelObject are used in connection with gels (bobs and sprites). For normal use it is only the last one that has interest.

The module GelObject contains two object definitions VSpriteObject and BobObject. The use of these object are best shown by an example.

Let first make a (virtual) sprite. This is done by defining an object:

```
STRUC VirtualSprite
  INHERIT VSpriteObject

  PROC Create
    Init
  ENDPROC Create

  // Colors
  DATA $0F00,$00F0,$000F

  // Sprite data
  DATA 12                                        // number of lines
  //   plane 0           plane 1
  DATA %1111111111111111,%1111111111111111  // 1. line
  DATA %1111111111111111,%1100000000000011
  DATA %1111111111111111,%1100000000000011
  DATA %1111000000001111,%1100111111110011
  DATA %1111000000001111,%1100111111110011
  DATA %1111000000001111,%1100110011110011
  DATA %1111000000001111,%1100110011110011
  DATA %1111000000001111,%1100111111110011
  DATA %1111000000001111,%1100111111110011
  DATA %1111111111111111,%1100000000000011
  DATA %1111111111111111,%1100000000000011
  DATA %1111111111111111,%1111111111111111  // 12. line
ENDSTRUC VirtualSprite
```

This object inherits the VSpriteObject and it contains a method Create and DATA statements defining the object.

To create a sprite with a shape defined by the DATA statements you have to execute a DIM statement:

```
DIM MySprite OF VirtualSprite
```

and then call the methods Create and Draw:

```
MySprite.Create
MySprite.Draw(50,50)
```

Now you can see the sprite on the screen. To move the sprite call the method Move (inherited from VSpriteObject), for instance

```
MySprite.Move(10,10)
```

At the end of your program the sprite has to be deleted:

```
MySprite.Delete
```

A bob is made in almost the same way. First you have to define a Bob structure:

```
STRUC Bob
  INHERIT BobObject

  PROC Create
    Init
  ENDPROC Create

  // Bob data
  DATA 1,9                 // Words per line and number of lines

  // Plane 0
  DATA %0000111111000011   // 1. line
  DATA %0011111111110011
  DATA %0011000011000011
  DATA %0000000000000000
  DATA %0000000000000000
  DATA %0000000000000000
  DATA %1100000000110011
  DATA %1111111111000000
  DATA %0011111100000011   // 9. line

  // Plane 1
  DATA %0000000000000000   // 1. line
  DATA %0000000000000000
  DATA %0000000000000000
```

```
    DATA %0011110000000011
    DATA %0011111111000011
    DATA %0000001111000011
    DATA %1100000000110011
    DATA %1111111111000000
    DATA %0011111100000011   // 9. line
  ENDSTRUC Bob
```

and execute a DIM statement:

```
  DIM MyBob OF Bob
```

The rest of the program is made in the same way as for virtual sprites.

You can have as many virtual sprite and bobs as you want on the screen. They are all made in the same way. Only the DATA statement may be different.

10.5 Memory.

This module is used to allocate memory from the Amiga's free memory pool.

The following routines are exported:

```
  FUNC malloc(BlockSize OF ULONG,MemType OF ULONG) OF ULONG
```

Allocate a memory block of size BlockSize and of type MemType (MEMF_PUBLIC, MEMF_CLEAR, MEMF_CHIP or MEMF_FAST).

The function returns the address of the block or zero if the block could not be allocated.

```
  PROC mfree(MemAdr OF ULONG)
```

Free a memory block previosly allocated by malloc

```
  PROC mfreeall
```

Free all memory blocks previosly allocated by malloc

The module contains a signal routine that calls mfreeall if the variables are cleared.

10.6 CodeManInclude and InterpreterInclude.

These modules contains definitions to be used to access Comal.CodeMan and Comal.Interpreter (see section V.2).

10.7 StartProgram.

This module exports one procedure:

```
PROC StartProgram(ProgramName$)

  Start a Comal program as a seperate process that will terminate
  itself. The parameter must be the full path of the program that must
  be stored as a code file.
```

10.9 Timer.

This module exports the following procedures:

```
PROC TimerEvent(TimeInterval,EventProc)

  Start the timer. The timer runs for the specified time interval (in
  seconds) and then you procedure EventProc is called. The event
  procedure
  is a procedure without parameters.


PROC AbortTimer(EventProc)

  Abort a time event before timeout.
```

The timer can be used to make periodic interrupts (see section 7.5). The timer interrupts the program at priority 20.

10.10 SerialComm.

This module is a serial communication module. The following procedures and functions are exported:

```
FUNC chars_in_buff(direction OF BYTE) OF SHORT

  This function  returns TRUE  if there are chracters in the IO buffer.
  Otherwise it returns FALSE.
```

If direction is 0 the input buffer is tested. Otherwise it is the output buffer (for the time being there is no output buffer and the return value with direction<>0 will always be FALSE).

PROC clear_buff(direction OF BYTE)

Clear the IO buffer. If direction is 0 the input buffer is tested. Otherwise it is the output buffer.

PROC ClearTermArray

Remove all break characters set by SetTermArray.

PROC CloseSerial

Close the serial device and make it available for other tasks.

FUNC OpenSerial OF BOOL

Open serial device. If success the value TRUE is returned. Otherwise FALSE is returned.

PROC receive(REF Data$,MaxChars OF LONG,MaxWait)

Read MaxChars characters into th string Data$. The procedure will return when all characters are received or MaxWait secs has elapsed.

PROC send(Data$)

Send the string Data$.

PROC set_mode(baud OF LONG,parity$,Len OF UBYTE,StopBits OF BYTE)

Set communication parameters. The parameters are:

```
baud:     The baud rate.
parity$:  "N" = no parity
          "E" = even parity
          "O" = odd parity
Len:      The number of bits in a word (6,7 or 8).
StopBits: The number of stop bits (1 or 2).
```

```
PROC SetTermArray(Arr OF TermArray)
```

Set termination characters. The type TermArray is defined in the module as

```
  TYPE TermArray=ARRAY(0..7) OF UBYTE
```

The actual parameter Arr should be filled with characters that is to break the reading. All 8 characters must be set in non increasing order. If less than 8 break characters is to be used (this is normal) fill the end of the array with the lowest value.

Example:

```
  DIM Array OF TermArray

    :

  Array():=$03    // ^C
  Array(0):=$1A   // ^Z
  Array(1):=$04   // ^D

  SetTermArray(Array())

    :
```

10.11 Operating system modules.

A number of modules are to be used to access the operating system routines. These routines are devided into two groups:

- Library interfaces for the system routines themself. These modules are mashine coded modules.
- Definitions necessary to call the system routines.

10.11.1 Library interface routines.

These modules are:

```
AslLibrary
DosLibrary
ExecLibrary
GadToolsLibrary
GraphicsLibrary
IntuitionLibrary
```

The modules contains interfaces for all the routines in the corresponding libraries. The names and the calling conventions are exactly the same as is used in a C program.

Example: This program opens a window and closes it again

```
  USE System
  USE IntuitionWindow
  USE IntuitionLibrary

  DIM NewWindow OF NewWindow
  DIM Window OF POINTER TO Window

  NewWindow.LeftEdge:=50
  NewWindow.TopEdge:=30
  NewWindow.Width:=300
  NewWindow.Height:=100
  NewWindow.BlockPen:=1
  NewWindow.Title:=ADR("Window demo")
  NewWindow.Screen:=ComalStruc@.IO_Screen
  NewWindow.Flags:=WINDOWDEPTH BITOR WINDOWDRAG
  NewWindow.Type:=CUSTOMSCREEN

  Window:=OpenWindow(ADR(NewWindow))
  WAIT 3
  CloseWindow(Window)
```

The library interface modules will not be described in detail here. Consult a description of the Amiga OS system routines.

10.11.2 Definitions for use in the library modules.

The definitions to be used in connection with calls to library routines are found in the following modules:

```
  AslInclude

    Structures:
      STRUC FileRequester
      STRUC FontRequester

    Types:
      TYPE FileReqPtr=POINTER TO FileRequester
```

```
  Constants:
    ASL_FuncFlag file requester tag values (FILF_DOWILDFUNC ..)
    ASL_ExtFlags1 file requester tag values (FILF1_NOFILES ..)
    ASL_FuncFlag font requester tag values  (FONF_FRONTCOLOR ...)
    requester types (ASL_FileRequest and ASL_FontRequest)
    tags for AllocAslRequestA and AslRequestA (ASL_Hail ...)


DosAslInclude

  Structures:
    STRUC DateStamp
    STRUC FileInfoBlock
    STRUC AChain
    STRUC AnchorPath

  Types:
    TYPE FileInfoPtr=POINTER TO FileInfoBlock
    TYPE AnchorPathPtr=POINTER TO AnchorPath


ExecLists

  Structures:
    STRUC Node
    STRUC List
      FUNC New CONSTRUCTOR
        Initializes the list.

  Constants:
    Node types (NT_TASK, NT_MESSAGE ...)


GadToolsInclude

  Structures:
    STRUC StringInfo
    STRUC Gadget
    STRUC NewGadget
    STRUC NewMenu

  Constants:
    Gadget types (GENERIC_KIND, BUTTON_KIND ...)
    IDCMP flags (ARROWIDCMP, BUTTONIDCMP ...)
    spacing constants (INTERWIDTH and INTERHEIGHT)
    NewGadget flags (NG_HIGHLABEL, PLACETEXT_LEFT ...)
    NewMenu types (NM_TITLE, NMITEM ...)
    return codes (GTMENU_TRIMMED ...)
    tags for tool kit functions (GTVI_NewWindow, GTVI_NWTags ...)
```

```
Gfx

  Functions and procedures:
    FUNC RASSIZE(w,h)     (in fact a macro in C include files)

  Structures:
    STRUC RectAngle
    STRUC BitMap


IDCMP

  Structures:
    STRUC IntuiMessage    (inherits Message from PortObjects)

  Constants:
    IDCMP classes (SIZEVERYFY, NEWSIZE ...)
    Mouse codes and key qualifiers (SELECTUP, ALTLEFT, ...)


IntuiText

  Structures:
    STRUC IntuiText
    STRUC TextAttr


  Constants:
    draw modes (JAM1, JAM2, XOR)


IoObjects

  Structures:

    STRUC IoRequest
      INHERIT Message
      PROC Init Constructor
        Initializes the length of the message.
      PROC Do
        Calls DoIO  in the  ExecLibrary with a pointer to the object as
        parameter.
      PROC Send
        Calls SendIO in the ExecLibrary with a pointer to the object as
        parameter.
      PROC Abort
        Calls AbortIO in the ExecLibrary with a pointer to the object as
        parameter.
```

```
    STRUC IoStdReq
      INHERIT IoRequest
      PROC Init Constructor
        Initializes the length of the message.


Layers

  Structures:
    STRUC Layer_Info

  Constants:
    Layer types etc. (LAYERSIMPLE, LAYERSMART ...)


MenuStrucs

  Structures:
    STRUC Menu
    STRUC MenuItem

  Constans:
    menu and item flags (MENUENABLED, CHECKIT, ITEMTEXT ...)
    MENUNULL


PortObjects

  Structures:
    STRUC MsgPort
      FUNC Init CONSTRUCTOR
      Makes the necessary initialization for a private port (allocates a
        signal bit etc.).
      PROC Delete DESTRUCTOR
        Removes the  port from  the list of public ports (if the method
        Add has been called) and deallocates the signal.
      PROC Add(REF Name$,Priority OF BYTE)
        Adds the port to the list of public ports.
      PROC Wait
      Wait for a message to arrive. It is possible to break the program
        during the wait.
      FUNC Get
        The ExecLibrary function GetMsg is called with a pointer to the
        port as parameter.
      PROC Put(REF Msg OF Message)
        The ExecLibrary procedure PutMsg  is called  with a  pointer to
        the port and the message as parameter.

    STRUC Message
      FUNC Init CONSTRUCTOR
```

```
      Initializes the message (but no reply port is created!!).
    PROC Wait
      Waits for the message to be replied and remove the message from
      the reply port.
    PROC Reply
      Replies the message.
```

Before you can use the message you will normally have to make a reply port and put an address of this port into the mn_ReplyPort field.

RastPort

Structures:
- STRUC AreaInfo
- STRUC TmpRas
- STRUC RastPort

Constants:
- drawing modes (JAM1, JAM2, COMPLEMENT, INVERSEVID)
- bits for RastPort flags (FRST_DOT, ONE_DOT, DBUFFER)

TagItem

Structures:
- STRUC TagItem

Constants:
- system tags (TAG_DONE, TAG_END, .. , TAG_USER)
- TAGFILTER_AND, TAGFILTER_NOT

View

Structures:
- STRUC ColorMap
- STRUC ViewPort
- STRUC View
- STRUC RasInfo

Constants:
- view modes (PFBA, DUALPF, HIRES, ...)

```
IntuitionWindow

  Structures:
    STRUC NewWindow
    STRUC Window

  Constants:
    flag bits set by Intuition (SCREENTYPE, WBENCHSCREEN, ...)
    window flags (WINDOWSIZING, WINDOWDRAG, ...)


IntuitionScreen

  Structures:
    STRUC NewScreen
    STRUC Screen
```

11 The Comal Intuition Tools (CIT).

******* NOTE! CIT requires Workbench 2.04 or higher *******

The Comal Intuition Tool (short CIT) is a set of modules that are used to build various Intuition items: screens, windows, gadgets etc.

Normally the creation of Intuition items involves filling out special structures with special values. If the item is used to act on response from a user (such as gadgets or menus) you also have to work with message ports,signals and messages. And you must evaluate each message to find out what to do.

By using CIT it is much easier. All the necessary structures are filled automatically with standard values that can be changed by calling object methods. You don't have to know anything about ports, signals and messages and the evaluation of a user action is done by the objects.

11.1 General description of CIT.

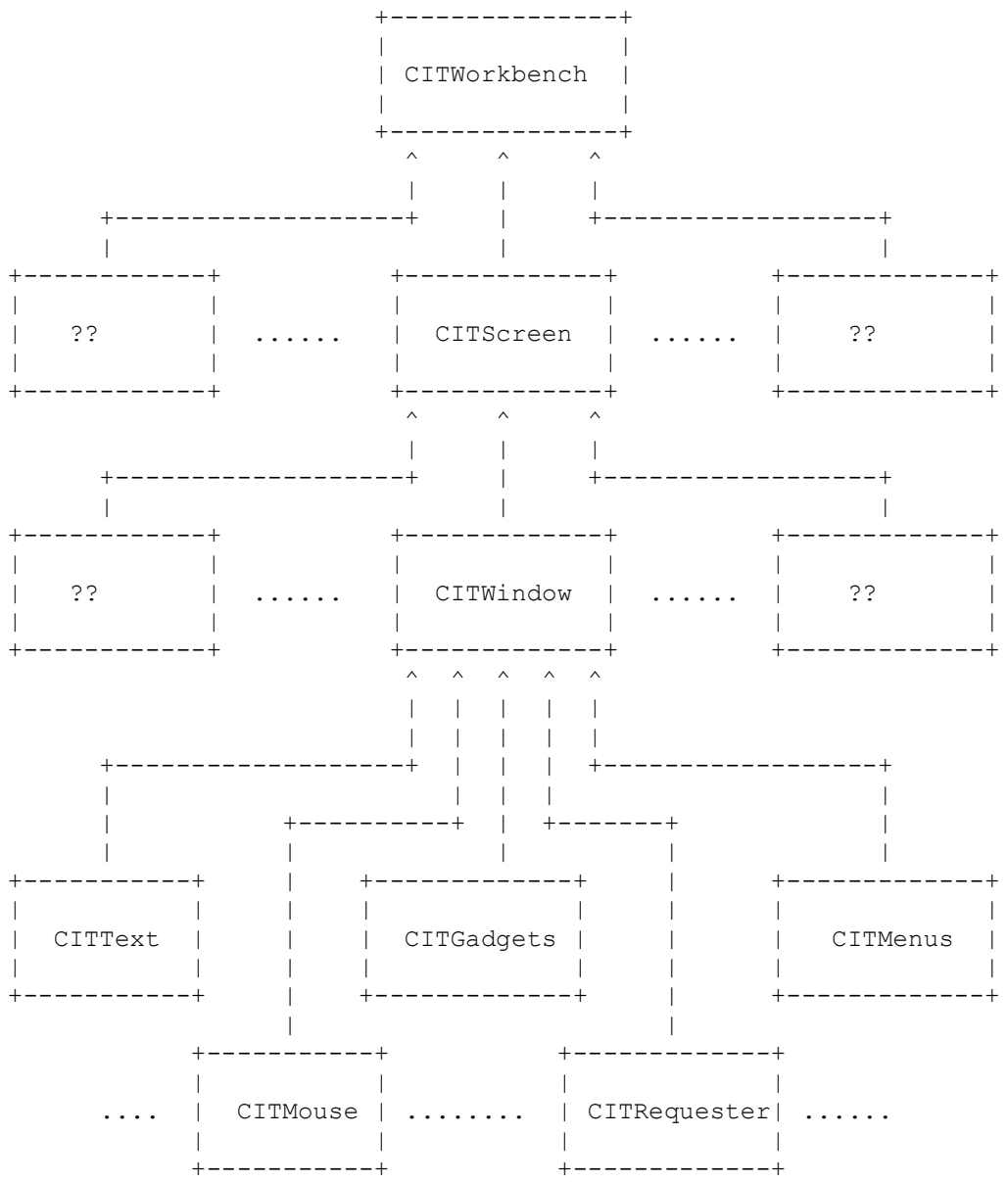
In CIT all Intuition items are treated as Objects. For example, a button gadget is an object that can be placed in a window and a window is an object that can be placed in a screen.

Certain objects have similar characteristics and can be classified into groups called classes. For instance, menus and gadgets are similiar in that they are placed in a window. They are classified as WindowClass objects.

In the same way a check box, a button gadget and a string input gadget are similar. They are classified as members of the class CITGadget (a subclass of the WindowClass).

All the objects are implemented as structures. You should think of the objects as physical items. First you have to create it. This is done by executing a DIM or LOCAL statement. Then you will proporly add some attributes like a position, a size or even a colour. This is done by calling methods in the structure. Finally you will place it somewhere. This is done by calling a method InsObject in the containing object.

As can be seen, all objects are placed in other objects. In this way objects form an object hierachy as shown below (the boxes containing '??' are not defined for the time being):



The use of all these different sorts of objects follow the same scheme:

To create an object you have to execute a DIM (or LOCAL) statement like:

```
DIM MyObject OF CITObject
```

Most of the visible objects have a Size method and a Position method to set its size and its position:

```
MyObject.Size(Width,Height)
MyObject.Position(LeftEdge,TopEdge)
```

and some have a Label method to connect a text to the object:

```
MyObject.Label(SomeText$)
```

These methods are always named Size, Position and Label, but sometimes there are additional parameters. In some of the gadgets, for instance, you have to specify if the text is to be placed inside the object itself or to the left, to the right etc.

If the object is to respond to a user action and you want immediate response, you have to call the method EventHandler with the name of a procedure to be called as the result of the users action:

```
MyObject.EventHandler(MyObjectEvent())
```

The number and types of the parameters in the event procedure are different from object type to object type (in the example we have indicated one parameter).

Having set all the attributes you have to insert it in another object. This is done by calling the method InsObject in the container object:

```
Container.InsObject(MyObject,Error)
```

When you are finished using the object you should remove it by calling the

method RemObject in the container object:

```
Container.RemObject(MyObject)
```

NOTE: Not all objects can be removed individually.

By removing the object MyObject all objects contained in this object will be removed, too.

Between the insertion and the removal you will normally be waiting for some termination code to take on a non zero value. This is typically done in a loop like:

```
WHILE NOT Terminate DO WAIT
```

While you are waiting all your event procedures will be called automatically.

In the following sections you will find a complete description of all the CIT objects.

11.2 Simple CIT examples.

As a simple example of the use of CIT let's make a window with a string input gadget and a button gadget to terminate the input.

First we have to load the modules we are using:

```
USE CITScreen
USE CITWindow
USE CITGadgets
```

and then we will create our window:

```
DIM DemoWindow OF CITWindow
```

give it the right size, place it on the correct position in the containing screen and make it active when it opens:

```
DemoWindow.Size(500,100)
DemoWindow.Position(60,50)
DemoWindow.Activate
```

We want to place the window in the ComalScreen which is a predefined screen found in the module CITScreen:

```
ComalScreen.InsObject(DemoWindow,Error)
```

Any error code is returned in the variable Error. We have to test this variable before we proceed:

```
IF Error THEN
  STOP "Could not create window"
ENDIF
```

The string input gadget is created, sized and positioned in the same way:

```
DIM NameInput OF StringGadget
NameInput.Size(300,15)
NameInput.Position(160,10)
```

We want a text to the left of the gadget:

```
NameInput.Label("Enter your name: ",LEFT)
```

This input gadget is placed in the window in exactly the same way as the window was placed in the screen:

```
DemoWindow.InsObject(NameInput,Error)
```

And now the button gadget:

```
DIM OkButton OF ButtonGadget
OkButton.Size(40,15)
OkButton.Position(10,-(6+15))
OkButton.Label("Ok",INSIDE)          // Place text inside the gadget
DemoWindow.InsObject(OkButton,Error)
```

Note that the y-coordinate of the position is negative. Then the gadget is positioned relative to the buttom border of the window.

The variable Error is only changed if an error occurs. Let's test it now:

```
IF Error THEN
  STOP "Could not create one or more gadgets"
ENDIF
```

Now we will sleep until the Ok button is pressed:

```
WHILE NOT OkButton.Pressed DO WAIT
```

The text typed into the input gadget is found in a field by name Value$ in the OkButton structure:

```
PRINT "Your name is: ",NameInput.Value$
```

As the final task we have to close the window (and all items inserted in the window):

```
ComalScreen.RemObject(DemoWindow)
```

The complete program looks like:

```
USE CITScreen
USE CITWindow
USE CITGadgets

DIM DemoWindow OF CITWindow
DemoWindow.Size(500,100)
DemoWindow.Position(60,50)
DemoWindow.Activate
ComalScreen.InsObject(DemoWindow,Error)
IF Error THEN
  STOP "Could not create window"
ENDIF

DIM NameInput OF StringGadget
NameInput.Size(300,15)
NameInput.Position(160,10)
NameInput.Label("Enter your name: ",LEFT)
DemoWindow.InsObject(NameInput,Error)

DIM OkButton OF ButtonGadget
OkButton.Size(40,15)
OkButton.Position(10,-(6+15))
OkButton.Label("Ok",INSIDE)
DemoWindow.InsObject(OkButton,Error)

IF Error THEN
  STOP "Could not create one or more gadgets"
ENDIF

WHILE NOT OkButton.Pressed DO WAIT

PRINT "Your name is: ",NameInput.Value$

ComalScreen.RemObject(DemoWindow)
```

In our next example we will add a Cancel button to the window and an event procedure to each of the button gadgets. These procedures are called each time the respective buttons are pressed. The OkEvent procedure is added in this way:

```
OkButton.EventHandler(OkEvent())

 :

PROC OkEvent(Id OF USHORT)
  Terminate:=1
ENDPROC OkEvent
```

The Cancel button (including the event procedure) is made in this way:

```
  DIM CancelButton OF ButtonGadget
  CancelButton.Size(60,15)
  CancelButton.Position(-(10+60),-(6+15))
  CancelButton.Label("Cancel",INSIDE)
  CancelButton.EventHandler(CancelEvent())
  DemoWindow.InsObject(CancelButton,Error)
  PROC CancelEvent(Id OF USHORT)
    Terminate:=2
  ENDPROC CancelEvent
```

Having done this and having testet for errors we will go to sleep. While sleeping the system will call the event procedures for you:

```
  WHILE NOT Terminate DO WAIT
```

The complete program looks like (note that the variable Terminate has to be dimensioned before you go to sleep):

```
  USE CITScreen
  USE CITWindow
  USE CITGadgets

  DIM Terminate OF SHORT

  DIM DemoWindow OF CITWindow
  DemoWindow.Size(500,100)
  DemoWindow.Position(60,50)
  DemoWindow.Activate
  ComalScreen.InsObject(DemoWindow,Error)
  IF Error THEN
```

```
  STOP "Could not create window"
ENDIF

DIM NameInput OF StringGadget
NameInput.Size(300,15)
NameInput.Position(160,10)
NameInput.Label("Enter your name: ",LEFT)
DemoWindow.InsObject(NameInput,Error)

DIM OkButton OF ButtonGadget
OkButtonSize(60,15)
OkButtonPosition(10,-(6+15))
OkButtonLabel("Ok",INSIDE)
OkButtonEventHandler(OkEvent())
DemoWindow.InsObject(OkButton,Error)
PROC OkEvent(id OF USHORT)
  Terminate:=1
ENDPROC OkEvent

DIM CancelButton OF ButtonGadget
CancelButton.Size(60,15)
CancelButton.Position(-(10+60),-(6+15))
CancelButton.Label("Cancel",INSIDE)
CancelButton.EventHandler(CancelEvent())
DemoWindow.InsObject(CancelButton,Error)
PROC CancelEvent(id OF USHORT)
  Terminate:=2
ENDPROC CancelEvent

IF Error THEN
  STOP "Could not create one or more gadgets"
ENDIF

WHILE NOT Terminate DO WAIT

CASE Terminate OF
WHEN 1
  PRINT "Hello ",NameInput.Value$,". Welcome to the world of CIT"
WHEN 2
  PRINT "You don't know your name!?"
ENDCASE

ComalScreen.RemObject(DemoWindow)
```

In the tutorial other examples are shown (chapter II.8-9).

11.3 CIT reference.

In the following subsections you will find a complete description of all the modules containing CIT objects, the object methods and the attributes that can be set in the objects.

11.3.1 CITWorkbench.

The CITWorkbench module containns the basic object CITWorkbench. You cannot create a CITWorkbench object yourself. It is already created in the module.

The following methods in CITWorkbench can be used:

```
PROC InsObject(REF WBObject OF WorkbenchClass,REF Error OF SHORT)
```

An object of the class WorkbenchClass is placed on the CITWorkbench. For the time being CITScreen is the only member of the Workbench-Class.

```
PROC RemObject(REF WBObject OF WorkbenchClass)
```

Remove an object from the the CITWorkbench.

If variables in the program are cleared all objects inserted in CITWorkbench are automatically removed.

11.3.2 CITScreen.

The object CITScreen found in the CITScreen module is member of the WorkbenchClass (the only member for the time being).

A CITScreen has to be created in a DIM (or LOCAL) statement and inserted in CITWorkbench. The screen will be a high resolution screen (640 pixels horizontal).

Example:

```
DIM MyScreen OF CITScreen
MyScreen.Label("My Own Screen")
MyScreen.Depth(3)
CITWorkbench.InsObject(MyScreen,Error)
```

The following methods in CITScreen can be used:

```
PROC InsObject(REF ScrObject OF ScreenClass,REF Error OF SHORT)
```

An object of the class ScreenClass is placed in a CITScreen object.

For the time being CITWindow is the only member of the ScreenClass.

```
PROC RemObject(REF ScrObject OF ScreenClass)
```

Remove a ScreenClass object from the CITScreen

```
PROC Label(t$)
```

Place a text in the drag bar of the screen. This method may be called only before the screen has been inserted in CITWorkbench.

```
PROC Depth(d OF SHORT)
```

Set the depth (the number of colors) of the screen. This method may be called only before the screen has been inserted in CITWorkbench.

```
PROC Lores
```

Make the screen a low resolution screen (the horizontal resolution is halfed). This method may be called only before the screen has been inserted in CITWorkbench.

```
PROC Interlace
```

Make the screen an interlace screen (the vertical resolution is doubled). This method may be called only before the screen has been inserted in CITWorkbench.

```
PROC ToBack
```

Send the screen behind all other screens. This method may be called only after the screen has been inserted in CITWorkbench.

```
PROC ToFront
```

Bring the screen in front of all other screens. This method may be called only after the screen has been inserted in CITWorkbench.

```
PROC Font(FontName$,FontHeight OF SHORT)
```

Set the default font of this screen. FontName$ is the name of the font (including the '.font' extension) and FontHeight is the desired height of the font.

If the font with the name and the specified height cannot be found nothing will happen.

Two predefined ready to use screens are exported from the CITSreen module: ComalScreen (the screen used by the Comal system) and WBScreen (the Workbench screen). They are used in the same way as other CITScreens (except that they cannot be inserted in or removed from CITWorkbench).

11.3.3 CITWindow.

In the module CITWindow the object CITWindow is found.

CITWindow is member of the ScreenClass (the only member for the time being). A CITWindow has to be created in a DIM (or LOCAL) statement and inserted in a screen.

Example:

```
DIM DemoWindow OF CITWindow
DemoWindow.Size(530,150)
ComalScreen.InsObject(DemoWindow,Error)
```

The following methods in CITWindow can be used:

```
PROC InsObject(REF WdObject OF WindowClass,REF Error OF SHORT)
```

An object of the class WindowClass is placed in the CITWindow.

All kinds of IDCMP events except MOUSEBUTTON and MOUSEMOVE events will be resend to the object as soon as they are recieved by the window.

```
PROC RemObject(REF WdObject OF WindowClass)
```

Remove a WindowClass object from the CITWindow.

```
PROC Position(x OF USHORT,y OF USHORT)
```

Set position of the window. This method may be called both before and after the window has been inserted in a screen.

```
PROC Size(w OF USHORT,h OF USHORT)
```

Set the size of the window. This method may be called both before and after the window has been inserted in a screen.

```
PROC Label(t$)
```

Set the window title. This method may be called both before and after the window has been inserted in a screen.

```
PROC Activate
```

Make window the active. This method may be called both before and after the window has been inserted in a screen.

```
PROC DepthGadget
```

Make a depth gadget in the upper right corner of the window. This method may be called only before the window has been inserted in a screen.

```
PROC SizingGadget
```

Make a sizing gadget in the lower right corner of the window. This method may be called only before the window has been inserted in a screen.

```
PROC DragBar
```

Add a drag bar to the window (so that the window can be moved by the user). This method may be called only before the window has been inserted in a screen.

```
PROC NoBorder
```

The window will be opened without a border. This method may be called only before the window has been inserted in a screen.

```
PROC InterruptPriority(n BYTE)
```

Set interrupt priority of window interrupts. This method may be called only before the window has been inserted in a screen

```
PROC ToFront
```

Bring the window in front of all other windows on the screen. This method may be called only after the window has been inserted in a screen.

```
PROC ToBack
```

Bring the window behind all other windows on the screen. This method

```
PROC CloseGadget
```

Make a close gadget in the upper left corner of the window. This method may be called only before the window has been inserted in a screen.

```
FUNC ClosePressed
```

This boolean function returns TRUE if the windows close gadget has been pressed. The status of the gadget will be set to FALSE after the call.

```
PROC CloseEventHandler(p OF CloseEventProc)
```

By calling this method with a procedure p, this procedure will be called immediately if the close gadget is pressed.

The parameter p is a procedure without parameters.

Example:

```
DemoWindow.CloseEventHandler(CloseEvent)

  :

PROC CloseEvent

  <do necessary actions>

ENDPROC CloseEvent
```

```
PROC SelectEventHandler(EventProc OF ButtonEventProc)
```

By calling this the method the procedure EventProc will be called when the left mouse button (select button) is pressed.

EventProc is an event procedure of the form:

```
PROC Button(Down OF BYTE,x OF SHORT,y OF SHORT)
```

At the call to the event procedure the parameter Down is TRUE if the button was pressed or FALSE if it was released. x and y are the the mouse position coordinates.

Example:

```
DemoWindow.SelectEventHandler(Button(,,))

  :

PROC Button(Down OF BYTE,x OF SHORT,y OF SHORT)
  <do anything>
ENDPROC Button
```

```
PROC MenuEventHandler(EventProc OF ButtonEventProc)
```

By calling this the method the procedure EventProc will be called when the right mouse button (menu button) is pressed.

For further description see SelectEventHandler.

Note that this method must be called before the window is inserted in the screen. Alse note that after calling this you cannot have menus attached to the window.

```
PROC PointerEventHandler(EventProc OF PointerEventProc)
```

If the method MouseMove has been called with the actual value TRUE the EventProc will be called when the mouse pointer is moved.

EventProc is an event procedure of the form:

```
PROC MousePos(x OF SHORT,y OF SHORT)
```

At the call to the event procedure the parameters x and y are the the mouse position coordinates.

Note that you have to respond to the event very fast.

PROC MouseMove(b OF SHORT)

Start mouse move event calling. As described above your event procedure will be called if the parameter b has the value TRUE.

PROC Coordinates(REF x OF SHORT,REF y OF SHORT)

This method will return the current position coordinates of the mouse pointer.

Example:

```
DemoWindow.EventHandler(MouseMove(,))
DemoWindow.MouseMove(TRUE)
  :
PROC MouseMove(x OF SHORT,y OF SHORT)
  PRINT AT 20,5: USING "x = -### ,  y = -###": x,y,
ENDPROC MouseMove
```

A predefined ready to use window ComalWindow is exported from the CIT-Window module. This is the standard IO window.

Only the methods InsObject, RemObject, ToFront and ToBack can be used with ComalWindow and the window cannot be removed from its containing screen. Otherwise the window is used in almost the same way as CITWindows.

11.3.4 CITBorder.

The module CITBorder contains the window class objects CITBorder.

The CITBorder is used to draw borders in the window and is a rather special object that you will probably never use. It is inherited by the CITView and CITText. It contains the following methods:

PROC Position(LeftEdge OF SHORT,TopEge OF SHORT)

Set the position of the upper left corner of the border. Like gadgets the position coordinates may be negative. The the position of the upper left corner will be placed relative to the right and bottom border of the window.

```
PROC Size(Width OF USHORT,Height OF USHORT,Direc OF USHORT)

  Set the size of the border. The parameter Direc specifies if the
  border should be raised (use the value UP), recessed (use the value
  DOWN) or if there should be no visible border (use the value
  NOBORDER).

  The default size of the border is the size of  the containing element
  (CITWindow or CITView).
```

The CITBorder is inherited by some other WindowClass objects (CITView, CITText and CITGraphics).

11.3.5 CITView.

The CITView object is a WindowClass object that at first sight looks like the CITBorder (which it iherits) but it is a very special WindowClass object in that any WindowClass object (including the CITView itself) can be inserted in (and sometimes removed from) a CITView.

A WindowClass object inserted in a CITView is placed relative to the border of the view.

The CITBorder contains the Size and Position methods inherited from CITBorder as well as InsObject and RemObject (see CITWindow for a description of these methods).

11.3.6 CITGadgets.

In this module you will find a number of different gadgets to be inserted in a window.

All these gadgets are member of the class GadgetClass. Members of this class have these methods (with some minor individual modifications) along with other special methods:

```
PROC Disable

  Disables the gadget so that the user cannot use it. Most of the
  gadgets will appear ghoasted.


PROC Enable

  Enables the gadget.
```

```
PROC Position(LeftEdge OF SHORT,TopEdge OF SHORT)

  Set the position (of the upper left corner of the  gadget) inside the
  window (or view). If inserted in a window the parameters LeftEdge and
  TopEdge are relative to the inner of a window (excluding borders).

  If the parameters (one ore both) are negative the position of the gad-
  gets upper left corner will be relative to the right/lower border of
  the window (view).


PROC Size(Width OF USHORT,Height OF USHORT)

  Set the width and height of the gadget.


PROC Label(text$,Flags OF ULONG)

  Set a text to be shown with the gadget. The Flags is  used to specify
  the position and the highlightning status of the text:

  The position values are

    LEFT        -   place text to the left of the gadget
    RIGHT       -   place text to the right of the gadget
    ABOVE       -   place text above the gadget
    BELOW       -   place text below the gadget
    INSIDE      -   place text inside the gadget


  These position values may be BITORed (or added) with

    HIGHLIGHT   -   show text highlighted (white)


PROC Font(Name$,Height OF SHORT)

  Set the name of the font to be used in the label. If the font cannot
  be found the default font (topaz80) will be used.


PROC Id(ID OF USHORT)

  Set an identification number. This number is for your use only.
```

```
PROC EventHandler(p OF GadEventProc)

  By calling this method with a procedure p, this procedure will be
  called immediately if  the gadget is activated (for input gadgets when
  you press the <ENTER> key).

  The parameter p is a procedure with one  USHORT parameter  (the iden-
  tifiction  number set by the method Id).
```

All the gadgets (except the TextGadget) have a current state. For all gadgets this state can be read in the field Value
(Value$ in the StringGadget).

Note that this Value field should never be changed directly. If it is possible to change it, there will be a method available for this purpose.

11.3.6.1 TextGadget.

The TextGadget is a special read only gadget. A TextGadget has a fixed label (set by the Label method) and a changeable text.

The following methods are special for the TextGadget:

```
PROC Text(t$)

  Set the  (changable) text.  The method  may be called both before and
  after the gadget has been inserted in a window.


PROC Border

  Call this method to get a recessed border around the text.
```

Example:

```
DIM Text OF TextGadget
Text.Position(214,5)
Text.Text("CIT demo")
DemoWindow.InsObject(Text,Error)
```

Since this gadget is a read only gadget it makes no sence to add an event procedure to this gadget.

11.3.6.2 ButtonGadget.

The well known button gadget adds the following method to all the common methods:

FUNC Pressed OF SHORT

This boolean function returns TRUE if the gadget has been pressed. The status of the gadget will be set to FALSE after the call.

11.3.6.3 CheckboxGadget.

The CheckboxGadget has the following methods:

PROC On

Set the state of the gadget to ON (the check mark will be shown). The method may be called both before and after the gadget has been inserted in a window.

PROC Off

Set the state of the gadget to OFF (the check mark will be removed). The method may be called both before and after the gadget has been inserted in a window.

11.3.6.4 StringGadget.

The StringGadget is used to enter a string of ASCII characters. This gadget has the following special method:

PROC Replace

Use this method to change the typing mode to the replace mode. Otherwise it is insert mode. The method may only be called before the gadget has been inserted in a window.

PROC TabCycle

Make the gadget a TabCycle gadget. If the user types Tab or Shift-Tab into a TabCycle gadget the next or previous TabCycle gadget will be activated. The method may only be called before the gadget has been inserted in a window.

```
PROC MaxChar(n OF USHORT)
```

This method sets the maximum number of input characters in the gadget. The default number is 64. The method may only be called before the gadget has been inserted in a window.

```
PROC Text(t$)
```

Set the content of the gadgets input field. The method may be called both before and after the gadget has been inserted in a window.

```
PROC Activate
```

Activate the gadget so that you can type in your text. The method may only be called after the gadget has been inserted in a window.

Example:

```
DIM StringGad OF StringGadget
StringGad.Label("Type in a text",LEFT)
StringGad.Position(150,48)
StringGad.Size(170,15)
DemoWindow.InsObject(StringGad,Error)
```

11.3.6.5 IntegerGadget.

The IntegerGadget is used to enter an integer number. This gadget has the following special method:

```
PROC Replace
```

Use this method to change the typing mode to the replace mode. Otherwise it is insert mode. The method may only be called before the gadget has been inserted in a window.

```
PROC TabCycle
```

Make the gadget a TabCycle gadget. If the user types Tab or Shift-Tab into a TabCycle gadget the next or previous TabCycle gadget will be activated. The method may only be called before the gadget has been inserted in a window.

```
PROC MaxChar(n OF USHORT)
```

This method sets the maximum number of input characters in the gadget. The default number is 64. The method may only be called before the gadget has been inserted in a window.

```
PROC Number(n OF ULONG)
```

Set the content of the gadgets input field. The method may be called both before and after the gadget has been inserted in a window.

```
PROC Activate
```

Activate the gadget so that you can type in your number. The method may only be called after the gadget has been inserted in a window.

11.3.6.6 SliderGadget.

A SliderGadget is a gadget used to show or control the amount of some quantity like a color, a volume or an intensity.

This gadget has the following methods:

```
PROC Limits(Min OF SHORT,Max OF SHORT)
```

The method is used to set the level limits, i.e. the minimum and maximum value of the quantity in question. The method may only be called before the gadget has been inserted in a window.

```
PROC Level(l OF SHORT)
```

The method is use to set the actual level of the quantity. The method may be called both before and after the gadget has been inserted in a window.

In the string used as parameter in the Label method (common for all gadgets) you may place the formatting characters '#' as in a PRINT USING statement. Then the current level will be printed.

Example:

```
DIM SliderGad OF SliderGadget
SliderGad.Position(150,88)
SliderGad.Limits(0,100)
SliderGad.Label("Fraction:###%",LEFT)
DemoWindow.InsObject(SliderGad,Error)
```

As a result of the line

```
SliderGad.Label("Fraction:###%",LEFT)
```

a text of the form

```
Fraction: 37%
```

will be printed on the left side of the gadget.

11.3.6.7 ScrollerGadget.

A ScrollerGadget is a gadget used to show and/or adjust the fraction and the position of a limited view into a larger area. The ScrollerGadget is a well known gadget. It is placed on the right side of the Comal editor window and it is used in all the drawer windows opened by Workbench

This gadget has the following methods:

```
PROC Top(t OF SHORT)

  Sets the first visible position in the area that the scroller
  represents. The method may be called both before and after the
  gadget has been inserted in a window.
```

```
PROC Total(t OF SHORT)

  Sets the total size of the area that the scroller represents.
  The method may be called both before and after the gadget has been
  inserted in a window.
```

```
PROC Visible(v OF SHORT)

  Sets the number of visible elements. The method may be called both
  before and after the gadget has been inserted in a window.
```

```
PROC Arrows(Size OF SHORT)

  Place arrows at the end of the scroller. The parameter is the width of
  each arrow button for a horizontal scroller or the height for a
  vertical scroller. The default size is 10. Use a size value  of zero
  to avoid arrows.

  The method  may only be called before the gadget has been inserted in
  a window.


PROC Orientation(Or OF SHORT)

  Set the orientation of the scroller, The parameter values are:

    HORIZONTAL    -   make a horizontal scroller
    VERTICAL      -   make a vertical scroller

  The default orientation is horizontal.

  The method may only be called before the gadget has  been inserted in
  a window.


The value found in the field Value is the current Top value.
```

Example:
Let's say we have a text of 137 lines (numbered 0 .. 136) and we can see 21 lines on the display (currently from line 32 to 52). The following lines will create a scroller representing the text:

```
DIM Scroler OF ScrollerGadget
Scroller.Position(500,5)
Scroller.Size(15,150)
Scroller.Total(137)             // A total of 137 lines
Scroller.Top(32)                // First visible (counting from zero)
Scroller.Visible(21)            // 21 visible lines
Scroller.Orientation(VERTICAL)
DemoWindow.InsObject(Scroller,Error)
```

11.3.6.8 CycleGadget.

A CycleGadget is used to allow the user to choose one among several choices. It appears as a raised rectangular button. A circular arrow glyph appears to the left and the current choice to the right. Clicking once on the gadget the next choice in a list will appear, while shift-clicking will show the previous choice.

This gadget has the following methods:

```
PROC Choices(REF Texts$())
```

The texts in the array Texts$() (all texts or up to the first empty text) is the choices that will appear in the gadget. These texts must be valid through the whole lifetime of the gaget.

The method may be called both before and after the gadget has been inserted in a window.

```
PROC Active(n OF SHORT)
```

Sets the choice with the ordinal number n (counting from zero) as the active choice in the gadget. The method may be called both before and after the gadget has been inserted in a window.

Example:

```
DIM CycleChoice$(4) OF 20
READ CycleChoice$()
DATA "Choice number 1"
DATA "Choice number 2"
DATA "Choice number 3"
DATA "Choice number 4"

DIM CycleGad OF CycleGadget
CycleGad.Position(330,8)
CycleGad.Size(190,14)
CycleGad.Label("Press here",LEFT)
CycleGad.Choices(CycleChoice$())
CycleGad.Active(2)                    // 'Choice number 3'
DemoWindow.InsObject(CycleGad,Error)
```

11.3.6.9 RadioButtonGadget.

Like the CycleGadget the RadioButtonGadget allow the user to choose one option from among several.

All the choices are shown as a text beside a small raised oval that looks like a small radio button. Exactly one one button is recessed and highlighted to indicate the selected choice.

This gadget has the following methods:

```
PROC Choices(REF Texts$(),Place OF ULONG)
```

The texts in the array Texts$() (all texts or up to the first empty text) is the choices that will appear. These texts must be valid through the whole lifetime of the gaget.

The parameter Place specifies where the text is to be placed. The values are:

```
LEFT  - place text on the left side of the buttons
RIGHT - place text on the right side of the buttons
```

The method may only be called before the gadget has been inserted in a window.

```
PROC Spacing(s OF SHORT)
```

Sets the amount of spacing (in pixels) between each button. The default value is zero (not very nice). The method may only be called before the gadget has been inserted in a window.

```
PROC Active(n OF SHORT)
```

Sets the choice with the ordinal number n (counting from zero) as the active choice in the gadget. The method may be called both before and after the gadget has been inserted in a window.

11.3.6.10 ListViewGadget.

A ListViewGadget, like the CycleGadget and the RadioButtonGadget, allows the user to select one among several choices. It consists of a scroller with arrows, an area where the choices are visible and an optional place where the current selection is shown. The user can browse through the list using the scroller or the arrows and may select an entry by clicking on that item.

This gadget has the following methods:

```
PROC ChoiceArray(REF Texts$())
```

The texts in the array Texts$() (all texts or up to the first empty text) is the choices that will appear in the gadget. These texts must be valid through the whole lifetime of the gaget.

The method may be called both before and after the gadget has been inserted in a window.

```
PROC ChoiceList(REF List OF List)
```

This is an alternative way of supplying the choices. The choices are put ito the ln_Name field of an Exec list.

The method may be called both before and after the gadget has been inserted in a window.

```
PROC Spacing(s OF USHORT)
```

Sets the amount of spacing (in pixels) between each entry in the list-view. The default value is zero. The method may only be called before the gadget has been inserted in a window.

```
PROC Top(n OF USHORT)
```

The ordinal number (counting from zero) of the top item visible in the listview. The method may be called both before and after the gadget has been inserted in a window.

```
PROC ScrollWidth(w OF USHORT)
```

Sets the width of the scroller used in the listview. The method may be called only before the gadget has been inserted in a window.

```
PROC ReadOnly
```

Make the listview a read-only listview. The method may be called only before the gadget has been inserted in a window.

```
PROC ShowSelected(REF StrgGad OF StringGadget)
```

Use this method to attach a StringGadget to the listview. The currently selected item will be shown in the gadget (ready to be edited). The parameter StrgGad must have been created (in a DIM or LOCAL statement) but not inserted in a window.

The method may be called only before the gadget has been inserted in a window.

```
PROC Selected(s OF USHORT)
```

Ordinal number (counting from zero) of the item to be placed into the display under the listview.

The method may be called both before and after the gadget has been inserted in a window. If called before without a prior call of the Show-Selected method, a read only TextGadget will be created.

Example:

```
DIM ListViewTexts$(4) OF 20
ListViewTexts$(1):="Choice1"
ListViewTexts$(2):="Choice2"
ListViewTexts$(3):="Choice3"
ListViewTexts$(4):="Choice4"

DIM ListView OF ListViewGadget
ListView.Position(200,128)
ListView.Size(100,40)
ListView.ChoiceArray(ListViewTexts$())
ListView.Selected(1)                    // A TextGadget is created
DemoWindow.InsObject(ListView,Error)
```

11.3.6.11 PaletteGadget.

The PaletteGadget lets the user pick a colour from a set of several. It consists of a number of coloured squares, one for each colour available.

This gadget has the following methods:

```
PROC Color(c OF UBYTE)
```

The selected colour of the palette. Default is zero.

The method may be called both before and after the gadget has been inserted in a window.

```
PROC IndicatorWidth(w)
```

Specifies the width of the current-colour indicator. The default width is 20 pixels.

The method may be called only before the gadget has been inserted in a window.

```
PROC IndicatorHeight(h)
```

This method asks for a current-colour indicator to be placed above the the colour selection squares. The height is specified as a parameter.

The method may be called only before the gadget has been inserted in a window.

```
PROC Depth(d OF SHORT)
```

Specifies the number of bitplanes that the gadget represents. Default is two. The method may be called only before the gadget has been inserted in a window.

```
PROC ColorOffset(Offset OF SHORT)
```

If there are more bitplanes in the screen than specified by the Depth method this is the number of the first colour to be displayed. Default is zero.

The method may be called only before the gadget has been inserted in a window.

Example:

```
DIM Palette OF PaletteGadget
Palette.Position(350,128)
Palette.Color(2)                      // Color 2 is the selected one
DemoWindow.InsObject(Palette,Error)
```

11.3.7 CITText.

The module CITText contains the window class object CITText.

The CITText is used to write texts in the window. It contains the same methods as CITBorder (it inherits CITBorder) as well as the following methods:

```
PROC PenColor(c OF SHORT)
```

Set the colour of the drawing pen.

```
PROC BackColor(c OF SHORT)

  Set the colour of the back ground pen. This value is only used if the
  method Transparent is called with the parameter value FALSE.


PROC Transparent(b OF SHORT)

  If b has the value TRUE the letters are drawn transparent, i.e. no
  back ground is drawn.


PROC Font(Name$,Height OF SHORT)

  Set the  name of the font to be used. If the font cannot be found the
  default font (topaz80) will be used.


PROC Print(x OF SHORT,y OF SHORT,t$)

  Print the text t$ at position x,y.
```

Example:

```
DIM Text OF CITText
Text.Position(100,50)
Text.Size(250,50,DOWN)
DemoWindow.InsObject(Text,Error)

Text.PenColor(3)
Text.Font("times.font",24)
Text.Print(10,10,"Hello world!")
```

11.3.8 CITGraphics.

CITGraphics in the module CITGraphics is a window class object containing a (very small) subset of the graphics routines of the Graphics module.

It contains the same methods as CITText (it inherits CITText) as well as the following methods:

```
PROC Clear

  Clear the graphics area defined by Position and Size.
```

```
PROC Color(c OF SHORT)
```

Set the color of the drawing pen.

```
PROC MoveTo(x,y)
```

Move (without drawing) the pen to the point with coordinates (x,y).

```
PROC DrawTo(x,y)
```

Draw a straight line from the current point to the point with coordinates (x,y).

```
PROC Plot(x,y)
```

Set a dot at the point with coordinates (x,y).

```
PROC DrawText(x,y,t$)
```

Print a text starting at the graphics coordinates (x,y). The Print method of CITText is used to actually print the text.

```
PROC Coordinates(Xmin,Xmax,Ymin,Ymax)
```

Define a new coordinate system. Xmin and Xmax will be the new x-coordinates of the left and right edge of the drawing area. Ymin will be the new y-coordinate of the bottom edge and Ymax wil be the y-coordinate of the top edge.

In CITGraphics all drawing are restricted to drawing area set by the Size method. This is also the case with DrawText.

11.3.9 CITMenus.

CITMenu in the module CITMenus is a window class object used to make menus in a window. It contains the following methods:

```
PROC Title(Label$,Flags OF USHORT,REF MenuId OF USHORT)
```

Set the title of a new menu. The parameters are:

Label$

The title (the headline) of a new series of menu items.

Flags

The only legal non zero value is

DISABLED

Disable this menu (all items and subitems) from start.
The menu can be enabled by calling the method On.

MenuId

An identification number is returned in this variable. The value is used in calls to the On and Off methods.

```
PROC Item(Label$,Key$,Flags OF USHORT,Exlude OF LONG,
                          Prc OF MenuEvent,REF MenuId OF USHORT)
```

Make a new menu item belonging to the menu created by the last call of the method Title. The parameters are:

Label$

The name of the menu item.

Key$

The first character in this string is used as a short cut for the menu item. If the string is the empty string there will be no short cut for this item.

Flags

This parameter can be zero or it can take on one or more of the following values:

DISABLED	Disable the item (and all subitems belonging to it) from start. The item can be enabled by calling the method On.
CHECKIT	A check mark will be drawn to the left of the title if the item is currently selected.
MENUTOGGLE	By using this value the user can toggle the check mark by selecting the item. Only to be used in connection with the CHECKIT flag value. Normally you will use this flag in connection with the CHECKIT flag or you will set an exclude mask.

CHECKED Make the item selected from start. Only to be used in connection with the CHECKIT flag value.

The values can be BITORed (or added) together.

Exclude

Set a 1 in a binary bitmask for each item in this (sub)menu you want to be deselected when the user selects this item. The right-most digit in the bitmask corresponds to the first item. Only to be used in connection with the CHECKIT flag value.

Prc

This is an event procedure with one unsigned short (USHORT) parameter. This procedure will be called when the item is selected. At the call the actual parameter value will be the menu number retur ned in the REF parameter MenuId. In this way you can identify the item if several items (or subitems) shares an event procedure.

MenuId

An identification number is returned in this variable. The value is used in calls to the On and Off methods.

```
PROC SubItem(Label$,Key$,Flags OF USHORT,Exlude OF LONG,
             Prc OF MenuEvent,REF MenuId OF USHORT)
```

Create a subitem belonging to the last item. The parameters are the same as described above (the Item method).

```
PROC Bar
```

Make a separator bar between the previous and the next (sub)item.

```
PROC On(MenuId OF USHORT)
```

Enable the menu or (sub)item identified by MenuId (the number returned by calls to the methods Title, Item and SubItem).

```
PROC Off(MenuId OF USHORT)
```

Disable the menu or (sub)item identified by MenuId (the number returned by calls to the methods Title, Item and SubItem).

Example:

This example shows part of a program (most of the event procedures are not shown) creates a menu in the standard IO window.

```
USE CITWindow
USE CITMenus

DIM Terminate OF SHORT

DIM Menu OF CITMenu
Menu.Title("Project",0,ProjectId)
Menu.Item("Open","O",0,0,PrcOpen(),OpenId)
Menu.Item("Save","",0,0,PrcSave(),SaveId)
Menu.Bar
Menu.Item("Print Mode","",0,0,PrcMode(),ModeId)
Menu.SubItem("Draft","",CHECKIT+CHECKED,%10,PrcMode(),DraftId)
Menu.SubItem("NLQ","",CHECKIT,%01,PrcMode(),NLQId)
Menu.Item("Print","P",0,0,PrcPrint(),PrintId)
Menu.Bar
Menu.Item("Quit","Q",0,0,PrcQuit(),QuitId)
Menu.Title("Edit",0,EditId)
Menu.Item("Cut","X",0,0,PrcCut(),CutId)
Menu.Item("Copy","C",0,0,PrcCopy(),CopyId)
Menu.Item("Paste","V",0,0,PrcPaste(),PasteId)
Menu.Bar
Menu.Item("Undo","Z",0,0,PrcUndo(),UndoId)
ComalWindow.InsObject(Menu,Error)

IF Error THEN
  PRINT "Could not create menu"
ELSE
  WHILE NOT Terminate DO WAIT
  ComalWindow.RemObject(Menu)
ENDIF

// *********** Event procedures ************

PROC PrcQuit(Num OF USHORT)
  Terminate:=TRUE
ENDPROC PrcQuit

PROC PrcOpen(Num OF USHORT)
  :
ENDPROC PrcOpen

PROC PrcSave(Num OF USHORT)
  :
```

11.3.10 CITRequester.

CITRequester in the module CITRequesters is a window class object used to make system requesters in a window. It contains only one method:

```
FUNC Request(Text$,YesNo$) OF LONG
```

The parameters are:

Text$: This is the body text of the requester.
This text may contain C-language formatting characters, or the new line character, CHR$(10).

YesNo$: The gadget text for one or more gadgets in the requester.
Each gadget text is separated by the chracter '|'.

The return value is zero if the rightmost gadget was pressed. Otherwise the return value is the number of the gadget (counting from the left).

Example:

```
DIM Rq OF CITRequester

  :

CASE Rq.Request("Text modified"10"Save text?","Yes|No|Cancel") OF
WHEN 0

  :

WHEN 1

  :

WHEN 2

  :

ENDCASE
```

11.4 Creating your own CIT classes.

In principle it is a simple task to make new CIT types belonging to one of the excisting CITClasses: WorkbenchClass, ScreenClasss, WindowClass etc.

But before we can do this it is necessary to know a little more about how CIT works.

All the CIT classes (except the base class CITWorkbench) contain two central methods:

```
FUNC CreateObject(REF Alfa OF CITAlfa, .. ) OF SHORT VIRTUAL

PROC DeleteObject VIRTUAL
```

and in addition all the WindowClasses contain an event method:

```
PROC HandleEvent(REF Msg OF IntuiMessage) VIRTUAL
```

The value of the first parameter in the CreateObject method is the container object into which this object is to be placed. This method may have some other parameters as indicated.

As can be seen in a lot of examples, an AlfaClass object is inserted in a CITAlfa object by executing a line of the form:

```
Alfa.InsObject(AlfaClassObject,Error)
```

The method InsObject now calls the method CreateObject in AlfaClassObject and it is in fact this method that does the actual insertion. After a succesfull return from CreateObject the container object Alfa inserts the new object in a book keeping list such that it knows which objects are currently inserted.

It can also be seen from the examples, that an AlfaClass object is removed from a CITAlfa object by executing a line of the form:

```
Alfa.RemObject(AlfaClassObject)
```

The method RemObject now calls the method DeleteObject in AlfaClassObject and after that removes it from its book keeping list. If AlfaClassObject itself contains inserted object the DeleteObject method in AlfaClassObject has to remove all its inserted objects (i.e. it calls the DeleteObject method in all the inserted object).

If an AlfaClass event occurs, the object Alfa traverses the book keeping list and calls the HandleEvent method in all the inserted objects one by one.

An important requiremet for this to work with many different objects is that the methods CreateObject, DeleteObject and HandleEvent are declared as virtual methods.

To make a new AlfaClass type you have to define a structure that inherits the base class AlfaClass or one of its ancestors. In this structure you will probably define new data fields and you are likely to overload one or more of the virtual methods CreateObject, DeleteObject and HandleEvent.

Here is a typical example:

```
STRUC ExtAlfa
  INHERIT AlfaClass

      :
  < some data fields >
      :

  FUNC CreateObject(REF Alfa OF CITAlfa) OF SHORT VIRTUAL

    // First create the ancestor object
    IF AlfaClass.CreateObject(Alfa) THEN

      < do other ExtAlfa specific creation >

      RETURN TRUE
    ELSE
      RETURN FALSE
    ENDIF

  ENDFUNC CreateObject

  PROC DeleteObject VIRTUAL

    < do ExtAlfa specific deletion >

    // Finally delete the ancestor object
    AlfaClass.DeleteObject

  ENDPROC DeleteObject

  < other methods >

ENDSTRUC ExtAlfa
```

Note that the base class AlfaClass defined in connection with the definition of the CITAlfa type is only a skeleton. It would not make much sence to create an object of this type (although it is possible). It is defined as follow:

{NEXT PAGE}

```
STRUC AlfaClass
  METHODTABLE

  DIM CITAlfa OF POINTER TO CITAlfa

  PROC Terminate DESTRUCTOR
    IF CITAlfa THEN
      LOCAL AlfaClass OF POINTER TO AlfaClass

      AlfaClass:=ADR(CITAlfa)-4
      CITAlfa@.RemObject(AlfaClass@)
    ENDIF
  ENDPROC Terminate

  FUNC CreateObject(REF Alfa OF CITAlfa, .. ) OF SHORT VIRTUAL
    CITAlfa:=ADR(Alfa)
    RETURN TRUE
  ENDFUNC CreateObject

  PROC DeleteObject VIRTUAL
    CITAlfa:=0
  ENDPROC DeleteObject

  PROC HandleEvent(REF Msg OF IntuiMessage) VIRTUAL
  ENDPROC HandleEvent

ENDSTRUC AlfaClass
```

The CreateObject method stores a pointer to the container object (you are free to use this in your own extensions) and DeleteObject deletes this pointer. The destructor Terminate removes the object if the object is going to loose its data area (at return from a procedure for instance).

The HandleEvent method is only present in the WindowClass.

You should look into the modules CITWorkbench, CITScreen and CITWindow to see the actual definitions (there are some minor differences from the general scheme) and to see what the content of the actual container object is.

Example:

As a complete example we will make an extension of the CITWin-dow (a member of the ScreenClass). This extension should have a fixed default position and size and an Ok-button in the lower left corner.

In addition there should be a method Wait. This method will wait until the Ok-button has been pressed.

The definition could be made in this way:

```
STRUC MyWindow
INHERIT CITWindow

USE CITGadgets
USE IntuitionScreen

DIM CloseButton OF ButtonGadget

FUNC Init CONSTRUCTOR
  NewWindow.LeftEdge:=50
  NewWindow.TopEdge:=30
  NewWindow.Width:=500
  NewWindow.Height:=150
  CloseButton.Size(60,14)
  CloseButton.Position(5,-(14+3))
  CloseButton.Label("STOP",INSIDE)
  RETURN TRUE
ENDFUNC Init

FUNC CreateObject(REF CITScr OF CITScreen0) OF SHORT VIRTUAL
  LOCAL Error OF SHORT

  IF CITWindow.CreateObject(CITScr) THEN
    InsObject(CloseButton,Error)
    IF Error THEN
      CITWindow.DeleteObject
      RETURN FALSE
    ELSE
      RETURN TRUE
    ENDIF
  ELSE
    RETURN FALSE
  ENDIF
ENDFUNC CreateObject

PROC Wait
  WHILE NOT CloseButton.Pressed DO WAIT
ENDPROC Wait

ENDSTRUC MyWindow
```

Note that there is no DeleteObject in the structure. The DeleteObject method in CITWindow will do the work.

You should look through the different CIT-modules to see other examples on definition of CIT classes.

V. MAKING EXECUTABLE PROGRAMS

There are two ways to make executable programs (i.e. programs that can be started by clicking on its icon) out of a Comal program text.

One way is to store the program as a code file. To execute the program the whole Comal system (the interpreter and all modules used in the program) must be present on the system disk.

Another way is to combine the interpreter, the program and all the modules used by the program into one executable file. This file can be executed without the presence of the Comal system.

1 Comal code file.

A Comal program can be stored on disk as a code file by using the menu item Save.. in the Programs menu.

By double clicking on the icon created along with the code file, the program is started (without loading the editor).

The program can be started from the Shell (CLI) by executing a command like:

```
Comal.Starter [parameters] ProgramName
```

Example:

```
Comal.Starter SCREEN=Workbench Bitplane=3 Programs/DemoProgram
```

The parameter SCREEN=Workbench is used by Comal.Starter. Both parameters can be read by the program.

The program is executed as a separate process and the current directory at start is the directory containing the code file.

The code file is much shorter than the combined file described in the next section and it is the best way to make an executable file if the Comal system is present.

2 Combined files.

The Combiner combines the interpreter, a program and all modules used by this program into one single file that can be executed independent of the Comal system.

The Combiner strips off the comments from the program and all the Comal modules and it strips off the information text from mashine coded modules in the SystemModules directory before it combines it.

The combined file can be executed by double clicking on the icon created by the Combiner or by starting it from the Shell (CLI) in the normal way (startup parameters are managed just like TOOLTYPE's put into the icon).

The current directory at start of the execution is the directory containing the file.

2.1 Starting the Combiner from the editor.

The simplest way to make a combined file is to start the Combiner from the editor by using the Combine.. menu item in the Programs menu. Before the Combiner is started a file requester will pop up, and you may select the name and destination of the combined file.

The current workspace size and stack size as well as the current state of Automatic Variables and Execute Window are used as default values for the combined file (may be changed by using the Tool Types in the icon).

If the editor uses the Workbench screen, the Tool Type

```
SCREEN=Workbench
```

is put into the icon of the combined file.

2.2 Starting the Combiner from Workbench.

The Combiner can be started from the Workbench. Activate the icon of the Combiner by clicking once on its icon, press the shift key and double click on the icon of a program (text or code).

The following Tool Types can be put into the icon of the Combiner or the icon of the program:

SCREEN

The only legal value is Workbench:

```
SCREEN=Workbench
```

The default value is a private screen.

AUTOVAR

This tool type is used to select the default AutoVar state of the combined file. The default value for the Combiner is On.

Example: AUTOVAR=Off

EXECWINDOW

This tool type is used to select the default ExecWindow state of the combined file. The default value for the Combiner is On.

WORKSPACE

Selects the size of the workspace used by the combined program. The default value is 75000 bytes.

Example: WORKSPACE=60000

STACK

Selects the size of the stack, if the combined program is started from Workbench. The default value is 8Kb.

Example: STACK=16000

If the combined program is started from the Shell, the stack size of the Shell will be used.

OUTPUT

Specify the path and/or the name of the ouput from the combiner.

Example: OUTPUT=Comal:Programs/
OUTPUT=ram:Hanoi

If the name ends with ':' or '/' the tool value is treated as a path and the name of the output will be the same as the input file name.

If no output file path is specified the name of the combined file will be the name of the input file with the extension '.cmb'.

2.3 Starting the Combiner from the Shell (CLI).

The Combiner may be started from the Shell. The command is:

```
Combiner [options] ProgramName
```

The options are the same as the Tool Types used if the Combiner is started from the Workbench.

Example:

```
Combiner OUTPUT=ram: WORKSPACE=50000 Demos/Polygon2
```

3 Using ToolTypes in executable files.

The field Tool Types inside the icon of an executable file (a code file or a combined file) is where you can set parameters for the program.

The TOOLTYPE's and the ToolValue's recognized by the system are described below. All TOOLTYPE's are transfered to the program.

SCREEN

By using the SCREEN tool type you select if Comal should open its own screen or use the workbench screen. The only legal value is Workbench.

```
SCREEN=Workbench
```

The default value is a private screen. But this screen will not be opened if EXECWINDOW is set to Off.

AUTOVAR

This tool type is used to select if the program should create variables automatically or if all variables has to be declared in a DIM or LOCAL statement before use.

Example: AUTOVAR=Off

EXECWINDOW

This tool type is used to select if the program should open the standard execute window (used by PRINT, INPUT etc.). The default value is On.

Example: EXECWINDOW=Off

WORKSPACE

Selects the size of the workspace used by the program. The default value is 75000 bytes.

Example: WORKSPACE=60000

EMPTY PAGE!

VI. DESCRIPTION OF THE Comal SYSTEM

The Comal system consists of several program files. During normal program devellopment three of these program files are used. These files are:

Comal

The editor

Comal.CodeMan

A program used to manipulate the program buffer. It generates intermediate code from ASCII lines and regenerate ASCII lines from intermediate code. It inserts new lines in the buffer and deletes lines from the buffer. And much more.

Comal.Interpreter

The interpreter that executes your programs.

The programs are executed as separate processes. The editor Comal communicates with Comal.CodeMan and Comal.Interpreter by sending messages.

1 The code manipulator Comal.CodeMan.

The program Comal.CodeMan is used to manipulate the program buffer. Once started it will stay as a resident process and all comumnication with the process goes through its message port Comal.CodeMan.

A message send to this port must be either a RexxMessage (explained in a later section) or it must be of the form:

```
STRUC CodeCommandMsg
  INHERIT Message
  DIM Command OF POINTER TO CodeCommand
ENDSTRUC CodeCommandMsg
```

where Message is a standard Exec message and CodeCommand is

```
STRUC CodeCommand
  DIM Cmd OF SHORT
  DIM Status OF SHORT
  DIM PrgBuf OF POINTER TO PrgBuf
  DIM Param1 OF ULONG
  DIM Param2 OF ULONG
  DIM CmdProc@ OF CmdProc
ENDSTRUC CodeCommand
```

Before a message is send to Comal.CodeMan the Cmd field is set to the command you want to be executed and the fields Param1 and Param2 is filled with corresponding command parameters.

Before Comal.CodeMan returns the message, it places the command status code in the field Status (zero is ok) and return values in the fields Param1 and Param2.

The command that can be executed are:

CODE_KILL (1) Kill CodeMan process

If there are no open program buffers the Comal.CodeMan process will be removed.

There are no return values from this command.

CODE_OPEN (2) Open program buffer

Create a new program buffer. If status is ok the fields PrgBuf and CmdProc is set. CmdProc can be used to call CodeMan directly without sending a message (considerably faster).

Command parameters: Param1 = program buffer length
Param2 = initial flags

Return values: None

CODE_CLOSE (3) Close program buffer

A program buffer previously opened by executing the command CODE_OPEN is closed.

Command parameters: None

Return values: None

CODE_CLEAR (4) Clear program buffer

Remove the current content of the program buffer.

Command parameters: None

Return values: None

CODE_LOAD (5) Read file into program buffer

Read program stored on disk in code form into the program buffer. The previous content of the program buffer is lost.

Command parameters: Param1 = Pointer to null terminated file name

Return values: Param1 = Line number
Param2 = Total number of lines in buffer

Note: Since Comal.CodeMan runs as a separate process (unless CmdProc is used) the complete path for the file must be specified.

CODE_SAVE (6) Write program buffer to file

The content of the program buffer is stored on disk in code form. The program buffer is unchanged.

Command parameters: Param1 = Pointer to null terminated file name

Return values: None

Note: Since Comal.CodeMan runs as a separate process (unless CmdProc is used) the complete path for the file must be specified.

CODE_GEN (7) Generate code for ASCII line

Intermediate code is generated for an ASCII program line. If an error occurs the return parameter Param1 points to the place where the error was detected.

Command parameters: Param1 = Pointer to null terminated program line
Param2 = Address of buffer for the code

Return values: Param1 = Pointer into program line
Param2 = Address of buffer for the code

Note: The buffer must be large enough to hold the code generated (500 bytes is recommented).

CODE_GENDIRECT (8)

Intermediate code is generated for an ASCII program line. An error is reported if the code generated cannot be executed directly (if the program line is part of a program structure). If an error occurs the return parameter Param1 points to the place where the error was detected.

```
  Command parameters:   Param1 = Pointer to null terminated program line
                        Param2 = Address of buffer for the code

  Return values:        Param1 = Pointer into program line
                        Param2 = Address of buffer for the code
```

Note:
The buffer must be large enough to hold the code generated (500 bytes is recommented).

```
CODE_REGEN (9)          Regenerate ASCII line

  Regenerate the current line (pointed at by the buffer pointer) into
  the corresponding ASCII line. This command can be viewed of as the
  reverse of CODE_GEN.

  Command parameters:   Param1 = ASCII buffer pointer

  Return values:        Param1 = Length of ASCII line generated
                        Param2 = Line number of current line

  Note: The generated ASCII line is null terminated and the length
        returned includes the terminating zero


CODE_INSERT (10)        Insert code line in buffer

  Insert intermediate code line in program buffer at the current
  position.

  Command parameters:   Param1 = Address of code

  Return values:        Param1 = Line number of current line
                        Param2 = Total number of lines in buffer


CODE_DELETE (11)        Delete current line in buffer

  The current line in buffer is deleted

  Command parameters:   None

  Return values:        Param1 = Line number of current line
                        Param2 = Total number of lines in buffer


CODE_REPLACE (12)       Replace line in buffer

  Replace the current line with the code given as parameter.  This is a
  combination of CODE_DELETE and CODE_INSERT.
```

```
  Command parameters:   Param1 = Address of code

  Return values:        Param1 = Line number of current line
                        Param2 = Total number of lines in buffer


CODE_UP (13)            Move pointer n lines up

  Move the buffer pointer up n lines (or to top of buffer).

  Command parameters:   Param1 = Number of lines to move up

  Return values:        Param1 = Line number of current line
                        Param2 = Total number of lines in buffer


CODE_DOWN (14)          Move pointer n lines down

  Move the buffer pointer down n lines (or to buttom of buffer).

  Command parameters:   Param1 = Number of lines to move down

  Return values:        Param1 = Line number of current line
                        Param2 = Total number of lines in buffer


CODE_BOB (15)           Move pointer to bottom of buffer

  Move the buffer pointer to bottom of buffer.

  Command parameters:   None

  Return values:        Param1 = Line number of current line
                        Param2 = Old line number


CODE_TOB (16)           Move pointer to top of buffer

  Move the buffer pointer to top of buffer.

  Command parameters:   None

  Return values:        Param1 = 1
                        Param2 = Old line number

CODE_SETADR (17)        Set pointer to specified address

  The code pointer is set to the line containing the specified address

  Command parameter:    Param1 = address
```

```
    Return values:        Param1 = Line number of current line
                          Param2 = Address offset in current line


  CODE_SETFLAGS (18)      Set flags

    Set the CodeMan flags

    Command parameters:   Param1 = New flag values of flags to chage
                          Param2 = Mask of flag bits to change

    Return values:        Param1 = new flag values
                          Param2 = old flag values


Flags:
  $0001:  List standard identifiers in capital
  $0002:  List in PC format


Status codes returned by CODE_OPEN:

  NOMEM (1)           Cannot allocate memory
  NUMINITERR (2)      IEEE initialization error


Status code  returned by other commands:

  NOTDIRECT (1)       Not direct command
  NOBUFSPACE (2)      Not enough room in program buffer
  ENDOFBUF (3)        Pointer at end of buffer
  ADDRNOTFOUND (4)    Address not found in programbuffer
  NOFILE (5)          File not found
  READERROR (6)       Error during reading
  WRITEERROR (7)      Error during writing
  ILLFORMAT (8)       Not legal Comal program file
  CODEMANINUSE (9)    Cannot kill CodeMan

All the structures and symbolic names are defined in the  module CodeMan-
Include.

The following  procedure CodeToAscii is an example of the use of the com-
mands. The procedure changes a code program file to an ASCII text file.


  PROC CodeToAscii(Name$,NewName$) CLOSED
    USE System
    USE InterpreterInclude
    USE ExecLists
```

```
USE PortObjects
USE ExecLibrary

DIM CodeManPort OF POINTER TO MsgPort
DIM CodeManMsg OF CodeCommandMsg
DIM ReplyPort OF MsgPort
DIM Command OF CodeCommand
DIM Buffer(500) OF UBYTE

CodeManMsg.mn_ReplyPort:=ADR(ReplyPort)

// Find CodeMan port
CodeManPort:=FindPort("Comal.CodeMan")
IF CodeManPort=0 THEN
  CleanUp
  STOP
ENDIF

// .. and open CodeMan
Command.Cmd:=CODE_OPEN
Command.Param1:=$8000
Command.Param2:=0
SendCommand(Command)
IF Command.Status THEN
  CleanUp
  STOP
ENDIF

Command.Cmd:=CODE_LOAD
Command.Param1:=ADR(Name$)
Command.Param2:=0
SendCommand(Command)
IF Command.Status THEN
  CleanUp
  STOP
ENDIF

OPEN FILE 1,NewName$,WRITE
REPEAT
  Command.Cmd:=CODE_REGEN
  Command.Param1:=ADR(Buffer())
  Command.Param2:=0
  SendCommand(Command)
  IF Command.Status THEN
    CleanUp
    STOP
  ENDIF
  PRINT FILE 1: CharArrayToString$(ADR(Buffer()))
  Command.Cmd:=CODE_DOWN
  Command.Param1:=1
```

```
    Command.Param2:=0
    SendCommand(Command)
  UNTIL Command.Status

  CleanUp

  PROC SendCommand(REF Command OF CodeCommand)
    CodeManMsg.Command:=ADR(Command)
    CodeManPort@.Put(CodeManMsg)
    CodeManMsg.Wait
  ENDPROC SendCommand

  PROC CleanUp
    IF Command.CmdProc THEN
      Command.Cmd:=CODE_CLOSE
      Command.CmdProc@(ADR(Command))
      Command.CmdProc:=0
    ENDIF
    CLOSE
  ENDPROC CleanUp

ENDPROC CodeToAscii
```

2 The interpreter Comal.Interpreter.

The program Comal.Interpreter is used to execute a program. Once started it will stay as a resident process and all communication with the process goes through message ports.

A message send to the port must be either a RexxMessage (explained in a later section) or it must be of the form:

```
STRUC InterpreterCommandMsg
  INHERIT Message
  DIM Command OF POINTER TO InterpreterCommand
ENDSTRUC InterpreterCommandMsg
```

where Message is a standard Exec message and InterpreterCommand is a structure of the form:

```
STRUC InterpreterCommand
  DIM Cmd OF SHORT           // Put your command in here
  DIM Status OF SHORT        // Look for command status here
  DIM Int_Port@ OF MsgPort   // Use this port in communication
  DIM Task@ OF Task          // Interpreter task pointer
  DIM BreakSigMask OF ULONG  // Signal used to stop running program
  DIM Param1 OF ULONG        // 1. parameter
  DIM Param2 OF ULONG        // 2. parameter
```

```
    DIM Param3 OF ULONG        // 3. parameter
    DIM Flags OF ULONG
  ENDSTRUC InterpreterCommand
```

Before a message is send to Comal.Interpreter the Cmd field is set to the command you want to be executed and the fields Param1 and Param2 is filled with corresponding command parameters. Before the message is returned, the command status code in the field Status (zero is ok) and return values in the fields Param1, Param2 and Param3 are set.

The command that can be executed are:

INT_KILL (1) Kill interpreter

If there are no active interpreters the master process Comal.Interpreter will be removed. A non zero status means that the master process cannot be closed (others are using it). There are no command parameters or return values.

INT_OPEN (2) Open interpreter

A new interpreter is opened. The new interpreter is in fact a new process and all the next commands are send to the port returned in the field Int_Port.

Command parameters: Param1 = address of open structure
Param2 = stack size of new process
Param3 = RUN flags (only if LOAD and RUN)

The open structure has the form:

```
  STRUC IntOpenStruc
    DIM Flags OF ULONG
    DIM WorkspaceLength OF ULONG
    DIM PrgId OF POINTER TO UBYTE
    DIM IO_Port OF POINTER TO MsgPort
    DIM Screen OF POINTER TO Screen
    DIM InterpreterId OF POINTER TO UBYTE
  ENDSTRUC IntOpenStruc
```

and the open flags are:

```
  INTOPEN_AUTOVAR  ($00000001)    Automatic creation of variables
  INTOPEN_COMMPORT ($00000002)    Create communication port
  INTOPEN_LOADPRG  ($20000000)    Open and load program.
  INTOPEN_OPENSCR  ($40000000)    Open separate screen
  INTOPEN_LOADRUN  ($A0000000)    Load and run as separate process
```

If flag bit 29 is set (LOADPRG and LOADRUN) the field PrgId is a pointer to a null terminated file name. Otherwise it is an address of a program buffer (returned by CODE_OPEN command to Comal.CodeMan).

return values: Param1 = address of communication port (if the INTOPEN_COMMPORT is set)

INT_CLOSE (3) Close program buffer

Close the interpreter and remove the process created by INT_OPEN. The master process Comal.Interpreter is not closed.

Command parameters: None

Return values: None

INT_SCAN (4) Scan program

Make a prepass scan of the program. The address returned can be send to CodeMan (CODE_SETADR) to get the exact error position.

Command parameters: None

Return value: Param1 = active program buffer
Param2 = pointer into program line (if error)

INT_RUN (5) Execute program

Execute program in main program buffer. The address returned can be send to CodeMan (CODE_SETADR) to get the exact error position. If the run flag RUNFLAG_STDIO ($00000001) is set, an execute window will be the standard IO device. Otherwise the IO_Port in the open structure is used.

Command parameters: Param1 = flags

Return values: Param1 = active program buffer
Param2 = pointer into program line (if error)
Param3 = address of return text

INT_COMMAND (6) Execute line as command

Execute a single code line as command.

Command parameters: Param1 = pointer to line

```
  Return values:        Param1 = active program buffer
                        Param2 = pointer into program line (if error)
                        Param3 = address of return text


INT_CONTINUE (7)        Continue

  Continue breaked program execution

  Command parameters:   None

  Return values:        None


INT_RETPRGBUF (8)       Return program buffer

  The commands INT_SCAN, INT_RUN, INT_COMMAND transfers the pro-
  gram buffer  to the interpreter and you are not allowed to change the
  buffer. To get the program buffer back execute this command.

  Command parameters:   None

  Return values:        None


INT_DISCARD (9)         Remove all modules

  Remove all modules loaded.

  Command parameters:   None

  Return values:        None


INT_LISTMODULES (10)    Return list of all modules

  A list of module structures for all modules are returned.

  Command parameters:   None

  Return values:        Param1 = address of first structure in list


INT_STARTTRACE (11)     Initiate tracing mode

  Scan program and start tracing. No lines are executed.

  Command parameters:   None
```

```
  Return values:        Param1 = active program buffer
                        Param2 = pointer into line (if error)
                        Param3 = return text


INT_STOPTRACE (12)      Stop tracing mode

  Command parameters:   None

  Return values:        None


INT_SINGLESTEP (13)     Execute one step

  Execute one step of program. The program must be in tracing mode.

  Command parameters:   None

  Return values:        Param1 = program buffer
                        Param2 = pointer to line (if error)
                        Param3 = return text


INT_LINESTEP (14)       Execute one line

  Execute one line of program. The program must be in tracing mode.

  Command parameters:   None

  Return values:        Param1 = program buffer
                        Param2 = pointer to line (if error)
                        Param3 = return text


INT_SETBREAK (15)       Set break point

  Set break point at specified line in program or module

  Command parameters:   Param1 = address of program buffer
                        Param2 = line number of break point line

  Return values:        None


INT_CLEARBREAK (16)     Clear break point

  Clear break point at specified line

  Command parameters:   Param1 = address of program buffer
                        Param2 = line number of break point line
```

```
    Return values:        None


  INT_CLEARALL (17)       Clear all break points

    Clear all break points in main program and modules.

    Command parameters:   None

    Return values:        None

Status codes returned by INT_OPEN:

  NOMEM (1)           Cannot allocate memory
  NUMINITERR (2)      IEEE initialization error


Status codes returned by all other commands are the same as is  send by a
running program.

The structures and all the symbolic names are defined in the module Inter-
preterInclude.

The following procedure StartProgram is an example of the use of the com-
mands. The procedure starts a program stored as a code program file.


  PROC StartProgram(PrgName$) CLOSED
    USE System
    USE InterpreterInclude
    USE ExecLists
    USE PortObjects
    USE ExecLibrary

    DIM IntCmd OF InterpreterCommand
    DIM IntCmdMsg OF InterpreterCommandMsg
    DIM ReplyPort@ OF MsgPort, ReplySigMask OF ULONG
    DIM MasterPort@ OF MsgPort
    DIM OpenStruc OF IntOpenStruc
    DIM RetSigMask OF ULONG
    DIM pos OF SHORT

    // Find Interpreter master port
    MasterPort:=FindPort(MasterPortName$)
    IF MasterPort=0 THEN
      STOP "Cannot find interpreter master"
    ENDIF
```

```
    // Remove path from program name
    pos:=LEN(PrgName$)+1
    REPEAT
      pos:-1
    UNTIL pos=0 OR PrgName$(pos..pos)=":" OR PrgName$(pos..pos)="/"
    Name$:=PrgName$(pos+1..)

    OpenStruc.Flags:=INTOPEN_LOADRUN BITOR INTOPEN_AUTOVAR
    OpenStruc.WorkspaceLength:=$8000
    OpenStruc.PrgId:=ADR(PrgName$)
    OpenStruc.Screen:=0                 // Use Workbench screen
    OpenStruc.IO_Port:=0                // No IO port
    OpenStruc.InterpreterId:=ADR(Name$) // Id of new process

    IntCmdMsg.mn_ReplyPort:=ADR(ReplyPort)
    IntCmdMsg.Command:=ADR(IntCmd)
    IntCmd.Cmd:=INT_OPEN
    IntCmd.Param1:=ADR(OpenStruc)
    IntCmd.Param2:=$4000                // Stack size
    IntCmd.Param3:=RUNFLAG_STDIO
    MasterPort@.Put(IntCmdMsg)
    IntCmdMsg.Wait

    IF IntCmd.Status<>0 THEN
      STOP "Cannot load program"
    ENDIF

    // The program is started - return

  ENDPROC StartProgram
```

3 The starter program Comal.Starter.

The default tool for a Comal code program file is the program Comal.Starter. This program loads the interpreter Comal.Interpreter (if not already loaded) and opens an interpreter process that reads the file and executes it as a stand alone process by using the open flags INTOPEN_LOADRUN (almost like the procedure StartProgram in section 2).

Having done this the Comal.Starter terminates.

VII. THE Comal AREXX INTERFACE

Each of the three main parts of the Comal system (Comal, Comal.CodeMan and Comal.Interpreter) has an AREXX interface. This chapter covers the commands supported and how to access them.

1 The editor AREXX interface.

For each project opened a separate process is started with its own AREXX port. The name of the port is Comal.Project.<project number>, for instance Comal.Project.001 and Comal.Project.002.

The AREXX commands accepted by the editor are (NOTE: for all commands the return code is RC_WARN (5) if the command is currently disabled):

NEW — Create a new project

Command format: NEW

Return values: The name of the AREXX port of the project

Return codes: RC_OK (0) if success
RC_ERROR (10) if the command failed

Example: NEW

CLEAR — Clear program buffer

Command format: CLEAR [FORCE]

Return values: None

Return codes: RC_OK (0) if success
RC_WARN (5) if user pressed the NO gadget

If the program in the program buffer has been changed the user will be prompted to accept unless the FORCE argument is specified.

Example: CLEAR FORCE

OPEN — Open and load a new project file

Command format: OPEN [FILENAME=<name>][MODE=ASCII|CODE][FORCE]

Return values: None

```
  Return codes:   RC_OK (0) if success
                  RC_WARN (5) if NO or CANCEL gadget pressed
                  RC_ERROR (10) if illegal or no file
```

If the file name is not specified the user is prompted for a file name via the file requester. The MODE argument specifies the file type (an ASCII text file or a Comal code file). The default mode is ASCII. If the program in the program buffer has been changed the user will be prompted to accept unless the FORCE argument is specified.

Example: `OPEN FILENAME=Demos/Integral1 FORCE`

SAVE — Save project

```
  Command format: SAVE [MODE=ASCII|CODE]

  Return values:  None

  Return codes:   RC_OK (0) if success
                  RC_WARN (5) if CANCEL gadget pressed
                  RC_ERROR (10) else
```

The MODE argument specifies the file type (an ASCII text file or a Comal code file). The default mode is ASCII. If the current project is unnamed the user is prompted for a file name via the file requester.

Example: `SAVE CODE`

SAVEAS — Save project under specified name

```
  Command format: OPEN [FILENAME=<name>] [MODE=ASCII|CODE]

  Return values:  None

  Return codes:   RC_OK (0) if success
                  RC_WARN (5) if NO or CANCEL gadget pressed
                  RC_ERROR (10) if illegal or no file
```

If the file name is not specified the user is prompted for a file name via the file requester. The MODE argument specifies the file type (an ASCII text file or a Comal code file).

Example: `SAVEAS FILENAME=Programs/Hanoi CODE`

PRINT Output the program to printer

Command format: PRINT

Return values: None

Return codes: RC_OK (0)

Example: PRINT

QUIT Terminate the project

Command format: QUIT [FORCE]

Return values: None

Return codes: RC_OK (0) if success
RC_WARN (5) if user pressed the NO gadget

If the program in the program buffer has been changed the user will be prompted to accept unless the FORCE argument is specified.

Example: QUIT

BLOCK Set block mode on/off

Command format: BLOCK

Return values: ON or OFF dependent on the block mode state after the command is executed.

Return codes: RC_OK (0)

Example: BLOCK

BLOCKSIZE Return the size of the current block

Command format: BLOCKSIZE

Return values: The returned number is in fact not the size of the block. The number of lines in the block is -RetVal+1 if RetVal<0 and RetVal+1 else. The number can be used directly as an argument in the LINE command to move to the other end of the block.

Return codes: RC_OK (0)

Example: BLOCKSIZE

CUT Move marked block to the clip board

Command format: CUT

Return values: None.

Return codes: RC_OK (0) if success
RC_ERROR (10) if out of memory

Example: CUT

COPY Move a copy of the marked block to the clip board

Command format: COPY

Return values: None.

Return codes: RC_OK (0) if success
RC_ERROR (10) if out of memory

Example: COPY

PASTE Insert content of clipboard into the program

Command format: PASTE

Return values: None.

Return codes: RC_OK (0) if success
RC_ERROR (10) if out of memory

Example: PASTE

ERASE Delete marked block

Command format: ERASE

Return values: None.

Return codes: RC_OK (0)

Example: ERASE

```
CURSOR            Move cursor

  Command format: CURSOR UP|DOWN|LEFT|RIGHT

  Return values:  None

  Return codes:   RC_OK (0) if success
                  RC_WARN (5) if border of text

  Example:         CURSOR DOWN


POSITION          Move cursor to specified position

  Command format: POSITION <place>  where place is:
                    SOF start of file
                    EOF end of file
                    SOL start of line
                    EOL end of line
                    SOV start of view (top of window)
                    EOV end of view (bottom of window)

  Return values:  None

  Return codes:   RC_OK (0)

  Example:         POSITION EOF


LINE              Move cursor up/down specified number of lines

  Command format: LINE <number>

  Return values:  None.

  Return codes:   RC_OK (0) if success
                  RC_WARN (5) if border of text

  Example:         LINE -5     /* Move 5 lines up  */


COLUMN            Move cursor left/right specified number of characters

  Command format: COLUMN <number>

  Return values:  None.

  Return codes:   RC_OK (0) if success
                  RC_WARN (5) if border of text
```

Example: COLUMN 5 /* Move 5 characters right */

TEXT Write the text argument at current cursor position

Command format: TEXT text

Return values: None.

Return codes: RC_OK (0)

Example: TEXT ' this text will be written'

NEWLINE Output a CR character

Command format: NEWLINE

Return values: None.

Return codes: RC_OK (0)

The command has the same effect as pressing the enter key.

Example: NEWLINE

INSLINE Insert en empty line at the cursor position

Command format: INSLINE

Return values: None.

Return codes: RC_OK (0)

Example: INSLINE

DELCHAR Delete character under cursor

Command format: DELCHAR

Return values: None.

Return codes: RC_OK (0)

Example: DELCHAR

```
DELLINE           Delete cursor line

  Command format: DELLINE

  Return values:  None.

  Return codes:   RC_OK (0)

  Example:        DELLINE


GETTEXT           Get the content of cursor line

  Command format: GETTEXT

  Return values:  The cursor line.

  Return codes:   RC_OK (0)

  Example:        GETTEXT


WRITESTATUS       Write text in status line

  Command format: WRITESTATUS text

  Return values:  None.

  Return codes:   RC_OK (0)

  Example:        WRITESTATUS 'this text will be written'


GETFILE           Get file from file requester

  Command format: GETFILE [FILENAME=<Name>][HAIL=<HailText>]

  Return values:  Selected file name (including full path).

  Return codes:   RC_OK (0) if success
                  RC_WARN (5) if CANCEL gadget presse

  The command will open a file requester. If a file name is specified
  this name will be written as the current file. The hail test will be
  written in the top of the file requester window.

  Example:        GETFILE FILENAME=System.mod HAIL=Select a module
```

INSERT Set insert mode

Command format: INSERT [ON|OF]

Return values: ON or OFF dependent on the insert mode state after the command is executed.

Return codes: RC_OK (0)

Flip insert mode or set insert mode to on or off.

Example: INSERT ON

ENTERINSERT Set enter mode

Command format: ENTERINSERT [ON|OF]

Return values: ON or OFF dependent on the enter mode state after the command is executed.

Return codes: RC_OK (0)

Flip enter mode or set enter mode to on or off.

Example: ENTERINSERT ON

2 The CodeMan AREXX interface.

The CodeMan AREXX port is named Comal.CodeMan. The AREXX commands accepted by the are:

KILL Remove the CodeMan process

Command format: KILL

Return values: None

Return codes: RC_OK (0) if success
RC_WARN (5) if the command failed

Example: KILL

```
OPEN              Create new program buffer

  Command format: OPEN [MEMORY=size]

  Return values:  An address which is used to identify the buffer.

  Return codes:   RC_OK (0) if success
                  RC_ERROR (10) if the command failed

  A new program buffer is created. If MEMORY  is not  specified the de-
  fault value $8000 will be used.

  Example:        OPEN MEMORY=25000


CLOSE             Close program buffer opened by the OPEN command.

  Command format: CLOSE ID=Id

  Return values:  None

  Return codes:   RC_OK (0) if success
                  RC_ERROR (10) if the program buffer was not found

  A  program  buffer  opened  by  the OPEN command is closed. The argu-
  ment Id is the address returned by the OPEN command.

  Example:        CLOSE ID=BUFFER


CLEAR             Clear program buffer

  Command format: CLEAR ID=Id

  Return values:  None

  Return codes:   RC_OK (0) if success
                  RC_ERROR (10) if the program buffer was not found

  The argument Id is the address returned by the OPEN command.

  Example:        CLEAR ID=BUFFER


LOAD              Load program file into program buffer

  Command format: LOAD ID=Id FILENAME=Name

  Return values:  None
```

Return codes: RC_OK (0) if success
RC_ERROR (10) if the buffer or file was not found

A program stored as a code file if loaded. The argument Id is the address returned by the OPEN command.

Example: LOAD ID=BUFFER FILENAME=MyFile

SAVE Save program buffer as a code program file.

Command format: SAVE ID=Id FILENAME=Name

Return values: None

Return codes: RC_OK (0) if success
RC_ERROR (10) if the no buffer or write error

The content of the program buffer is stored as a code file. The argument Id is the address returned by the OPEN command.

Example: SAVE ID=BUFFER FILENAME=MyFile

MAKECODE Generate intermediate code

Command format: MAKECODE ID=Id ProgramLine

Return values: Address of the generated code.

Return codes: RC_OK (0) if success
RC_ERROR (10) else

Intermediate code for the ASCII program line specified as argument is generated. The address of the generated code is returned.

Example: MAKECODE ID=BUFFER 'IF Alfa=7 THEN'

MAKEDIREC Generate intermediate code for direct command

Command format: MAKEDIREC ID=Id ProgramLine

Return values: Address of the generated code.

Return codes: RC_OK (0) if success
RC_ERROR (10) else

Intermediate code for the command specified as argument is generated. The address of the generated code is returned.

Example: MAKEDIREC ID=BUFFER 'PRINT sin(1)'

MAKEASCII Change intermediate code to ASCII

Command format: MAKECODE ID=Id

Return values: Address of the generated ASCII line.

Return codes: RC_OK (0) if success
RC_ERROR (10) else

An ASCII text line for the current buffer line is generated. The address of the generated line is returned.

Example: MAKEASCII ID=BUFFER

INSERT Insert code in the program buffer

Command format: INSERT ID=Id

Return values: None.

Return codes: RC_OK (0) if success
RC_ERROR (10) else

The code generated by the last MAKECODE or MAKEDIREC is inserted at the current position of the buffer pointer.

Example: INSERT ID=BUFFER

DELETE Delete the current buffer line

Command format: DELETE ID=Id

Return values: None

Return codes: RC_OK (0) if success
RC_ERROR (10) else

Example: DELETE ID=BUFFER

REPLACE Replace code in the program buffer

Command format: REPLACE ID=Id

```
  Return values:  None.

  Return codes:   RC_OK (0) if success
                  RC_ERROR (10) else

  The current line in the program buffer is replaced by the code genera-
  ted by the last MAKECODE or MAKEDIREC.

  Example:        REPLACE ID=BUFFER


POSITION          Move buffer pointer to specified position

  Command format: POSITION ID=Id <place>  where place is:
                    SOF start of file
                    EOF end of file

  Return values:  None

  Return codes:   RC_OK (0) if success
                  RC_ERROR (10) else

  Example.        POSITION ID=BUFFER SOF


LINE              Move buffer pointer up/down specified number of lines

  Command format: LINE ID=Id <number>

  Return values:  None.

  Return codes:   RC_OK (0) if success
                  RC_WARN (5) if border of text
                  RC_ERROR (10) else

  Example:        LINE ID=BUFFER -5 /* Move 5 lines down  */
```

3 The interpreter AREXX interface.

For each interpreter opened a separate process is started with its own AREXX port. The name of the port is Comal.Interpreter.<id>. For the interpreters opened by the editor the <Id> if 001, 002 etc. For instance Comal.Interpreter.001.

Only two AREXX commands are accepted by the interpreter:

```
KILL              Remove the Interpreter master

  Command format: KILL

  Return values:  None

  Return codes:   RC_OK (0) if success
                  RC_WARN (5) if the command failed

  Example:         KILL


EXECUTE           Load and execute program file.

  Command format:
      EXECUTE FILENAME=Name [WORKSPACE=Size] [STACK=Size]

  Return values:  None

  Return codes:   RC_OK (0) if success
                  RC_WARN (10) else

  A program  stored on  disk in  code form  is loaded  and executed. If
  WORKSPACE is  not specified  the default size $8000 is used. If STACK
  is not specified the default stack size $4000 is used.

  Example:        EXECUTE FILENAME=Comal:Programs/GraphDemo
```

4 An AREXX script example.

The following script will cut off a marked block and save it on disk as a code file. This is done by opening a separate program buffer and move the block into this buffer line by line.

```
/* Save block as code file  */

OPTIONS RESULTS
'BLOCKSIZE'
IF RC > 0
  THEN DO
    OPTIONS
    WRITESTATUS 'No block marked'
  END
  ELSE DO
    BlockSize = RESULT
    IF ARG() = 0
    THEN DO
```

```
    'GETFILE'
    File = RESULT
  END
  ELSE DO
    File = ARG(1)
  END
  IF RC = 0
  THEN DO
    'BLOCK'
    IF BlockSize < 0
    THEN DO
      OPTIONS
      'LINE' BlockSize
      BlockLen = -BlockSize+1;
      BlockSize =0
    END
    ELSE DO
      BlockLen = BlockSize+1;
      BlockSize = -BlockSize
    END
    address 'Comal.CodeMan'
    OPTIONS RESULTS
    'OPEN'
    ID = RESULT
    DO BlockLen
      address
      OPTIONS RESULTS
      'GETTEXT'
      Line = RESULT
      'CURSOR DOWN'
      address
      OPTIONS
      MAKECODE 'ID ' ID Line
      INSERT 'ID ' ID
      'LINE ID ' ID 1
    END
    'SAVE ID ' ID FILENAME File
    'CLOSE' 'ID ' ID
    address
    'LINE' BlockSize
  END
END
```

VIII. COMAL IO DEVICES

1 What is a Comal device?

A Comal IO device is a sort of a file with a predefined name. The name of a Comal device always ends with a colon (:). At the start of the Comal system three Comal devices are defined:

```
"ds:"   the standard IO window
"kb:"   the keyboard attached the the IO window
"lp:"   the printer
```

These device names can be used in a OPEN FILE satements and a SELECT statements.

Example:

```
OPEN FILE 1,"ds:",WRITE
SELECT OUTPUT "lp:"
```

The devices are primariy intented for use in SELECT statements. At the start of a program execution implicite

```
SELECT OUTPUT "ds:"
```

and

```
SELECT INPUT "kb:"
```

are executed.

2 Making new devices.

It is possible to add new devices or to replace one of the predefined devices. A new device is added by calling the internal Comal procedure Add-ComalDevice (found in the module SystemCode).

All devices are linked together in a list and AddComalDevice links the new device into the start of this list. Thus adding a device having the same name as an excisting device will in fact replace that device.

A device is removed from the list by calling RemComalDevice.

The procedure AddComalDevice is called with an initialized device structure that completely describes the device:

```
  STRUC IoDevice
    DIM NextDevice OF POINTER TO IoDevice
    DIM Name OF ULONG
    DIM Type OF USHORT
    DIM Reserved OF SHORT
    DIM Open OF ULONG
    DIM Close OF ULONG
    DIM Read OF ULONG
    DIM Write OF ULONG
    DIM ReadLn OF ULONG
    DIM WriteLn OF ULONG
    DIM Scan OF ULONG
    DIM GetStrmPtr OF ULONG
    DIM SetStrmPtr OF ULONG
    DIM StreamErr OF ULONG
  ENDSTRUC IoDevice
```

Field desription:

NextDevice

A pointer to the next device in the list of devices. Set by the Comal system.

Name

A pointer to the name of the device. Must end with a colon (:).

Type

The possible device types are:

```
  SEQ_DEVICE (0)  sequential device (for instance a printer)
  CRT_DEVICE (1)  used by windows
  KBD_DEVICE (2)  keyboard
  RBF_DEVICE (3)  random access device
```

Open open device

An address of an open procedure of the form:

```
  FUNC Open(Name OF ULONG,Mode OF USHORT,REF Eof OF SHORT)
```

where

Name is a null terminated string of characters. The start of the

string is the device name. The characters after the colon are transfered from the name in the OPEN or SELECT statement and can be used to specify special parameters for the device (for instance: SELECT OUTPUT "sp: baud=1200").

Mode is one or more of the following BITORed together:

ACCESS_READ (1)	read only device (such as a keyboard)
ACCESS_WRITE (2)	write only device (such as a printer)
ACCESS_NEW (4)	delete excisting (if window for instance)

At return the function sets the parameter Eof (end of file) to TRUE or FALSE and returns an ULONG which is used to indentify the device (zero means error).

Close close the device

An address of a procedure of the form:

```
PROC Close(Id OF ULONG)
```

The procedure is called when a CLOSE statement is executed or when another device is selected in a SELECT statement. Id is the identifier returned by the open procedure.

The procedure should not necessarily close the device. If for instance the device is a CRT_DEVICE like "ds:" it is not smart to close the window each time another device is selected by the SELECT statement.

Read read a block of bytes

An address of a function of the form:

```
FUNC Read(Id,Data,REF MaxBytes,REF BreakMask) OF SHORT
```

where

Id OF ULONG is the device identifier returned by OPEN.
Data OF ULONG is the address of a read buffer.
MaxByte OF LONG is the number of bytes to read.
BreakMask OF ULONG is a mask of signals that should break the reading. Not all devices supports this facility.

Before return the routine must set the actual number of bytes read in MaxBytes and then return -1 (if error), 0 if OK or 1 if end of stream was reached.

```
Write         write a block of bytes

  An address of a function of the form

    FUNC Write(Id,Data,Length) OF SHORT

  where

    Id OF ULONG is the device identifier returned by OPEN.
    Data OF ULONG is the address of the data to write.
    Length OF LONG is the number of bytes to write.

  Before return the routine must set the actual number of bytes written
  in Length and then return TRUE (1) if OK or FALSE (0) if error.

  If this field is set for a KBD_DEVICE, guide texts in INPUT statements
  are sent to this function.


ReadLn        read one line

  An address of a function of the form

    FUNC ReadLn(Id,Data,REF MaxLength) OF SHORT

  where

    Id OF ULONG is the device identifier returned by OPEN.
    Data OF ULONG is the address of a read buffer.
    MaxByte OF LONG is the maximal number of bytes to read.

  The routine reads characters until a line end character is read or
  Max-Bytes characters are read. The routine is responsible for editing
  during the input (if CRT_DEVICE).

  Before return the routine must set the actual number of bytes read in
  MaxBytes (The line end character not included) and then return -1 (if
  error), 0 if OK or 1 if end of stream was reached.


WriteLn         write one line

  An address of a function of the form

    FUNC WriteLn(Id,Line,Length) OF SHORT

  where

    Id OF ULONG is the device identifier returned by OPEN.
    Line OF ULONG is the address of the line to write.
    Length OF LONG is the length of the line to write.

  The routine must terminate by writing a line termination character to
  the device. Before return the routines must set the actual  number of
  bytes written  in Length  and then return TRUE (1) if OK or FALSE (0)
  if error.
```

```
Scan          scan the device

An address of a function of the form

  FUNC Scan(Id,Data,REF Length) OF SHORT

where

  Id OF ULONG is the device identifier returned by OPEN.
  Data OF ULONG is the address of a read buffer.
  Length OF LONG is the length of the read buffer.

The function should test if there are "characters" ready in the
device. If this is the case ot should return the first "character".
Otherwise it should return without reading  any characters.  A
"character"  may be more than  one byte.  A keyboard  for instance
returns more than one byte if a function key is pressed.

Before return the routine must set the actual number of bytes read in
Length and  then return  -1 (if error), 0 if OK or 1 if end of stream
was reached.


Get           get position of the stream pointer

Get is the address of a function. The format of this function depends
on the device type.

  CRT_DEVICE:   FUNC Get(Id,REF Row, REF Col) OF SHORT

    where

      Id OF ULONG is the device identifier.
      Row OF SHORT is used to return row number of the cursor
      Col OF SHORT is used to return column number of the cursor

  RBF_DEVICE:   FUNC Get(Id,REF Offset) OF SHORT

    where

      Id OF ULONG is the device identifier.
      Offset  OF  LONG  is  used  to return the current offset of the
      stream pointer.

Other devices should set these fields to zero. For both functions the
return value is TRUE if operation succeded and otherwise FALSE.
```

```
Set           set position of the stream pointer

Set is the address of a function. The format of this function depends
on the device type.

  CRT_DEVICE:

      FUNC Set(Id,Row,Col) OF SHORT

    where

      Id OF ULONG is the device identifier.
      Row OF SHORT is the new row number of the cursor
      Col OF SHORT is the new column number of the cursor


  RBF_DEVICE:

      FUNC Get(Id,REF Offset) OF SHORT

    where

      Id OF ULONG is the device identifier.
      Offset OF LONG is the new offset of the stream pointer.

Other devices should set these fields to zero. For both functions the
return value is TRUE if operation succeded and otherwise FALSE.


StreamErr

For the time being not used. Set to zero.


The fields  Name, Type,  Open, Close,  Get (CRT and RBF) and Set (CRT and
RBF) must be set. Other fields may be set to zero if the device  does not
support the function.


The following module adds a serial device "sp:" to the list of Comal
devices:


{NEXT PAGE}
```

```
// Serial device "sp:"

MODULE SerialDevice

  USE System
  USE SystemCode
  USE PortObjects
  USE IoObjects
  USE ExecLibrary

  STRUC IOTArray
    DIM TermArray0 OF ULONG
    DIM TermArray1 OF ULONG
  ENDSTRUC IOTArray

  STRUC IOExtSer  // Standard IO request to serial.device
    INHERIT IOStdReq             // Standard IO-request
    DIM io_CtrlChar OF ULONG     // xON, xOFF, INQ, ACK
    DIM io_RBufLen OF ULONG      // length of read buffer
    DIM io_ExtFlags OF ULONG     // Not used
    DIM io_Baud OF ULONG         // Baud rate
    DIM io_BrkTime OF ULONG      // duration of break signal in micros
    DIM io_TermArray OF IOTArray // Termination characters
    DIM io_ReadLen OF UBYTE      // Bits per read character
    DIM io_WriteLen OF UBYTE     // Bits per write character
    DIM io_StopBits OF UBYTE     // Number of stop bits
    DIM io_SerFlags OF UBYTE     // Flags set by serial device
    DIM io_Status OF USHORT
  ENDSTRUC IOExtSer

  STRUC IoDevice
    DIM NextDevice OF POINTER TO IoDevice
    DIM Name OF ULONG       // Pointer to device name 'sp:'
    DIM Type OF USHORT      // Type of device
    DIM Reserved OF SHORT
    DIM Open OF ULONG
    DIM Close OF ULONG
    DIM Read OF ULONG
    DIM Write OF ULONG
    DIM ReadLn OF ULONG
    DIM WriteLn OF ULONG
    DIM Scan OF ULONG
    DIM GetStrmPtr OF ULONG
    DIM SetStrmPtr OF ULONG
    DIM StreamErr OF ULONG
  ENDSTRUC IoDevice

  DIM CMD_READ OF SHORT
  DIM CMD_WRITE OF SHORT
  DIM CMD_NONSTD OF SHORT
```

```
CMD_READ:=2
CMD_WRITE:=3
CMD_NONSTD:=9

DIM SerReq OF POINTER TO IOExtSer
DIM ReplyPort OF POINTER TO MsgPort
DIM DevOpen OF BOOL
DIM SerDevice OF IoDevice
DIM SerDevName$ OF 3

SerDevName$:="sp:"
SerDevice.Name:=ADR(SerDevName$)
SerDevice.Type:=0         // Sequential device
SerDevice.Open:=ADR(SerOpen(,,))
SerDevice.Close:=ADR(SerClose())
SerDevice.Read:=ADR(SerRead(,,,))
SerDevice.Write:=ADR(SerWrite(,,))
SerDevice.ReadLn:=ADR(SerReadLn(,,))
SerDevice.WriteLn:=ADR(SerWriteLn(,,))
AddComalDevice(ADR(SerDevice))

FUNC SerOpen(Name OF ULONG,Mode OF USHORT,REF Eof OF SHORT)
                                                    OF ULONG
  ALLOCATE(SerReq,MEMF_PUBLIC)
  ALLOCATE(ReplyPort,MEMF_PUBLIC)
  SerReq@.mn_ReplyPort:=ReplyPort

  // Open serial.device'
  IF OpenDevice("serial.device",0,SerReq,0) THEN
    DEALLOCATE(SerReq)
    RETURN 0
  ENDIF
  DevOpen:=TRUE

  // Set TermArray
  SerReq@.io_Command:=CMD_NONSTD+2
  SerReq@.io_TermArray.TermArray0:=$1C0A0A0A
  SerReq@.io_TermArray.TermArray1:=$0A0A0A0A
  SerReq@.Do  // Set TermArray

  Eof:=FALSE
  RETURN $FFFFFFFF
ENDFUNC SerOpen

PROC SerClose(Id OF ULONG)
  IF DevOpen THEN
    SerReq@.Abort
    dummy:=SerReq@.mn_ReplyPort@.Get
    CloseDevice(SerReq)
```

```
    DEALLOCATE(SerReq)
    DEALLOCATE(ReplyPort)
    DevOpen:=FALSE
  ENDIF
ENDPROC SerClose

FUNC SerRead(Id OF ULONG,Buffer OF ULONG,REF MaxLen OF,
             LONG,BreakMask OF ULONG) OF SHORT
  SerReq@.io_SerFlags:=SerReq@.io_SerFlags BITAND %10111111
  SerReq@.io_Command:=CMD_READ
  SerReq@.io_Length:=MaxLen
  SerReq@.io_Data:=Buffer
  SerReq@.Send               // Read data
  SerReq@.Wait
  MaxLen:=SerReq@.io_Actual  // Return number of bytes read
ENDFUNC SerRead

FUNC SerWrite(Id OF ULONG,Data OF ULONG,REF Length OF LONG)
                                                      OF BOOL
  SerReq@.io_Command:=CMD_WRITE
  SerReq@.io_Length:=Length
  SerReq@.io_Data:=Data
  SerReq@.Send               // Write data
  SerReq@.Wait
  IF SerReq@.io_Error THEN
    RETURN FALSE
  ELSE
    RETURN TRUE
  ENDIF
ENDFUNC SerWrite

FUNC SerReadLn(Id OF ULONG,Buffer OF ULONG,REF MaxLen OF LONG)
                                                        OF SHORT
  LOCAL Ptr OF POINTER TO UBYTE

  SerReq@.io_SerFlags:=SerReq@.io_SerFlags BITOR %01000000
  SerReq@.io_Command:=CMD_READ
  SerReq@.io_Length:=MaxLen
  SerReq@.io_Data:=Buffer
  SerReq@.Send                // Read data
  SerReq@.Wait
  IF SerReq@.io_Error THEN
    RETURN -1
  ENDIF
  Ptr:=ADR(Buffer)+SerReq@.io_Actual-1
  CASE Ptr@ OF
  WHEN $1C
    MaxLen:=SerReq@.io_Actual-1
    RETURN 1        // End of file
  WHEN $0A
```

```
      MaxLen:=SerReq@.io_Actual-1
      RETURN 0
    OTHERWISE
      MaxLen:=SerReq@.io_Actual
      RETURN 0
    ENDCASE
  ENDFUNC SerReadLn

  FUNC SerWriteLn(Id OF ULONG,Data OF ULONG,REF Length OF LONG)
                                                          OF BOOL
    LOCAL ch OF UBYTE

    SerReq@.io_Command:=CMD_WRITE
    SerReq@.io_Length:=Length
    SerReq@.io_Data:=Data
    SerReq@.Send               // Write data
    SerReq@.Wait
    IF SerReq@.io_Error THEN
      RETURN FALSE
    ENDIF
    ch:=$0A
    SerReq@.io_Command:=CMD_WRITE
    SerReq@.io_Length:=1
    SerReq@.io_Data:=ADR(ch)
    SerReq@.Do     // Write LF
    IF SerReq@.io_Error THEN
      RETURN FALSE
    ELSE
      RETURN TRUE
    ENDIF
  ENDFUNC SerWriteLn

  PROC SerSignal(s OF LONG) SIGNAL
    CASE s OF
    WHEN SIG_CLOSE,SIG_DISCARD
      RemComalDevice(ADR(SerDevice))
    OTHERWISE
      // No action
    ENDCASE
  ENDPROC SerSignal

ENDMODULE SerialDevice
```

3 Making your own IO window.

It is possible to replace the standard IO window opened by the interpreter by your own window. To see how this can be done it is necessary to know what is happening before a program is started.

When the interpreter receives an execute command the IO is redirected to the IO port (see section V.2) and the standard console device closes the execute window (if it is open). The console device closes the window when the signal SIG_CLEAR is send.

Then the program is scanned and during this scanning process all modules are loaded and are initialized by executing the initialization part of the module. A module can use this initialization to add a new "ds:" device and a new "kb:" device.

After the scanning an implicite SELECT OUTPUT "ds:" and an implicite SELECT INPUT "kb:" is executed. If a module has added a new "ds:" device and a new "kb:" device these devices are put into the start of the device list and it will be these devices that are used by the SELECT statements.

As a consequence your IO window will be opened and the standard IO window will not be used.

In the directory ModuleDev you will find an example of such a module programmed in C (IoWindow.c and IoWindow.interface).

IX. MACHINE CODED MODULES

Comal has been designed to be a modular language. It is possible to devide a project into several modules. These modules can be written in as different languages as assembler, C as well as the Comal language itself.

Seen from the users of the modules there are no difference in the modules wether they are written in assembler, C or Comal. The routines in the modules are called in exactly the same way. Very often a module is first written in Comal and then (if necessary) it is rewritten in C or even assembler.

In chapter III.9 development of modules in Comal was described. In this chapter machine coded modules (written in C or assembler) will be described.

1.0 The format of a mashine coded module.

A mashine coded module consists of three parts:

- the interface part
- the initialization and signal routines
- the routine part (procedures and functions in the module)

In the following sections 1.1-1.3 the three parts are described. C programmers do not need to read these sections.

1.1 The interface part.

In the interface part of a module all necessary informations about the exported routines are stored (the name and the address of the routine, the type of the routine and its parameters etc.).

The interface part is a table of elements of the form (described in not 100% correct C):

```
struct NameTable NameTable[NumNames];
```

where the NameTable structure is defined as

```
struct NameTable
{
  void *RoutineAddress;   /* The address of the routine           */
  struct ProcType *Type;  /* The address of the type descriptor   */
  char *RoutineName;      /* A pointer to the name of the routine */
}
```

The information of the type of the routine is stored in a structure Proc-Type. Routines with the same type can use the same structure.

The structure ProcType has the form:

```
struct Param
{
  UBYTE Flags;          /* Set to 0x80 if REF parameter       */
  BYTE  SecondaryType;  /* Set to zero                        */
  WORD  PrimaryType;    /* Parameter type (se include files)  */
};

struct ProcType
{
  WORD TypeId;          /* PROC_ID or FUNC_ID                 */
  WORD ReturnType;      /* Return type (FUNC) or zero (PROC)  */
  WORD *TypeDescriptor; /* Set to zero                        */
  UWORD Flags;
  UBYTE StackUse1;      /* Stack use of type descriptors      */
  UBYTE StackUse0;      /* Primary parameter stack use        */
  UWORD NumPar;         /* Number of formal parameters        */
  struct Param Param[NumPar];
};
```

Set Flags to 0x0001 if it is a string function returning a C string (null terminated ASCII string).

The stack use are the number of bytes used to transfer the parameters on the stack. The secondary stack is used to transfer the address of a type descriptor for structured types (ARRAY, FUNC and PROC) and REF parameters.

1.2 The initialization and signal routines.

The initialization routine is the very first mashine instructions in the module. It is a function with the C format:

```
short Init(struct ComalStruc *ComalStruc,struct Module *Module)
```

This means that the addresses of the ComalStruc and the module structure for the module in question are pushed onto the stack (as two long words) before a call is made to the first instruction of the module.

The initialization routine should as a minimum fill the module structure with the address of the name table and the number of names in the name table and then return the status code (in D0.W). Zero is the OK code.

Very often an initialization routine also stores the address of the Comal-Struc for later use.

A signal routine is a procedure with the C format:

```
void signal(short SignalNum)
```

To tell the comal system that there is a signal routine, the address of the routine must be placed in the module structure by the initialization routine. Otherwise there are no difference between a signal routine in a Comal module and a mashine coded module. The same signal numbers are send.

All routines, including the initialization routine and the signal routine, may use registers D0-D1/A0/A1. The content of all other registers must be restored before a return.

1.3 The routine part.

Procedures and functions that are to be exported from the module (those referenced in the name table) must have the form used by most C compilers (for instance SAS/C ver. 5.xx).

When a procedure or a function is called from Comal, the actual parameters are pushed onto the stack.

Float value parameters are pushed as two long words. Value parameters for the integer types are pushed as one long word (byte and short integers are sign extended before they are pushed).

Text values are transfered as an address to the text value that is null terminated.

Parameters of the type STRUC, ARRAY, PROC and FUNC as well as REF parameters of any kind are pushed as addresses.

Values returned by functions are passed back in register D0 (D0 and D1 if float).

All routines may use registers D0-D1/A0/A1. The content of all other registers must be restored before a return.

2.0 An assembler programmed module.

Let us as an example show how a module is programmed in assembler. We are going to make a module containing the following procedures and functions:

```
  FUNC Even(i OF LONG) OF SHORT
  FUNC Hex$(i OF ULONG)
  PROC Capital(REF t$)
```

The interface part contains a name table of three elements

```
NameTable:      DC.L    CapitalName       * Name of procedure
                DC.L    CapitalType       * Type descriptor
                DC.L    Capital           * Routine address

                DC.L    HexName
                DC.L    HexType
                DC.L    Hex

                DC.L    EvenName
                DC.L    EvenType
                DC.L    Even

CapitalName:    DC.B    'Capital',0
HexName:        DC.B    'Hex$',0
EvenName:       DC.B    'Even',0
```

and type descriptors for each function/procedure:

```
CapitalType:    DC.W    ProcTypeId        * Procedure
                DC.W    0                 * No return value
                DC.L    0                 * Always set to zero
                DC.W    $0000             * No flags
                DC.B    4                 * Sec. stack (REF)
                DC.B    4                 * Address = 4 bytes
                DC.W    1                 * One parameter
                DC.B    $80               * REF parameter
                DC.B    0                 * Always zero
                DC.W    StringTypeId      * Parameter type

HexType:        DC.W    FuncTypeId        * Function
                DC.W    StringTypeId      * Return value
                DC.L    0
                DC.W    $0001             * C string is returned
                DC.B    0                 * No sec. stack use
                DC.B    4                 * Int paramter = 4 bytes
                DC.W    1                 * One parameter
                DC.B    $00               * Value parameter
                DC.B    0
                DC.W    UlongTypeId       * Parameter type
```

```
EvenType:       DC.W    FuncTypeId        * Function
                DC.W    ShortTypeId       * Return type
                DC.L    0
                DC.W    $0000             * No flags
                DC.B    0                 * No sec. stack use
                DC.B    4                 * Int parameter = 4 bytes
                DC.W    1                 * One parameter
                DC.B    $00               * Value parameter
                DC.B    0
                DC.W    LongTypeId        * Parameter type
```

The initialization routine is very simple:

```
Init:     MOVE.L  4(A7),ComalStruc          * Store ComalStruc addr
          MOVE.L  8(A7),A0
          MOVE.L  #NameTable,MS_Names(A0)   * Name table address
          MOVE.W  #3,MS_NumName(A0)         * Number of names
          MOVEQ   #0,D0                     * Initialization OK
          RTS
```

The complete source for the module looks like:

```
; Module Demo

          include 'Comal.i'

          SECTION DemoCode,CODE
          CNOP    0,2

Start:    MOVE.L  4(A7),ComalStruc          * Store ComalStruc addr
          MOVE.L  8(A7),A0
          MOVE.L  #NameTable,MS_Names(A0)   * Name table address
          MOVE.W  #3,MS_NumName(A0)         * Number of names
          MOVEQ   #0,D0                     * Initialization OK
          RTS

Capital:  MOVE.L  4(A7),A1                  * Get string address
CapLoop:  MOVE.B  (A1)+,D0                  * Get next character
          BEQ.S   CapEnd                    * Return if zero
          CMP.B   #'a',D0                   * Small letter?
          BCS     CapLoop                   * No -> loop
          CMP.B   #'z',D0
          BHI     CapLoop
          BCLR    #5,-1(A1)                 * Change to capital
          BRA     CapLoop
CapEnd:   RTS
```

```
Hex:      MOVE.L  ComalStruc,A0             * Get address of ComalStruc
          MOVE.L  CS_CurrWorkBottom(A0),A1  * .. and work space start
          MOVE.L  A1,D0                     * Set as return value
          LEA     10(A1),A1                 * Reserve 10 bytes
          MOVE.L  A1,CS_CurrWorkBottom(A0)  * Set new bottom
          MOVE.L  D0,A1                     * Get start of string
          MOVE.B  #'$',(A1)+                * Start with hex specifier
          MOVE.B  #'0',(A1)+
          MOVE.L  4(A7),D1                  * Number to D1
          BEQ.S   HexEnd                    * Return if zero
          MOVEM.L D2/D3,-(A7)               * Store registers
          SUBQ.L  #1,A1                     * Point after $
          MOVEQ   #8,D3
          LEA     Digits(PC),A0
HxLoop1:  ROL.L   #4,D1                     * Find first non zero digit
          SUBQ.W  #1,D3
          MOVE.W  D1,D2
          AND.W   #$000F,D2
          BEQ     HxLoop1
HxLoop2:  MOVE.B  0(A0,D2.W),(A1)+          * Nex hex digit to buffer
          ROL.L   #4,D1
          MOVE.W  D1,D2
          AND.W   #$000F,D2
          DBRA    D3,HxLoop2
          MOVEM.L (A7)+,D2/D3
HexEnd:   CLR.B   (A1)                      * Terminate string
          RTS                               * .. and return

Digits:   DC.B    '0123456789ABCDEF'

Even:     MOVE.L  4(A7),D0
          ADDQ.L  #1,D0
          AND.L   #1,D0
          RTS


          SECTION DemoData,DATA
          CNOP    0,2

CapitalType:   DC.W    ProcTypeId         * Procedure
               DC.W    0                  * No return value
               DC.L    0                  * Always set to zero
               DC.W    $0000              * No flags
               DC.B    4                  * Sec. stack (REF)
               DC.B    4                  * Address = 4 bytes
               DC.W    1                  * One parameter
               DC.B    $80                * REF parameter
               DC.B    0                  * Always zero
               DC.W    StringTypeId       * Parameter type
```

```
HexType:        DC.W    FuncTypeId        * Function
                DC.W    StringTypeId      * Return value
                DC.L    0
                DC.W    $0001             * C string is returned
                DC.B    0                 * No sec. stack use
                DC.B    4                 * Int paramter = 4 bytes
                DC.W    1                 * One parameter
                DC.B    $00               * Value parameter
                DC.B    0
                DC.W    UlongTypeId       * Parameter type

EvenType:       DC.W    FuncTypeId        * Function
                DC.W    ShortTypeId       * Return type
                DC.L    0
                DC.W    $0000             * No flags
                DC.B    0                 * No sec. stack use
                DC.B    4                 * Int parameter = 4 bytes
                DC.W    1                 * One parameter
                DC.B    $00               * Value parameter
                DC.B    0
                DC.W    LongTypeId        * Parameter type


NameTable:      DC.L    CapitalName
                DC.L    CapitalType
                DC.L    Capital

                DC.L    HexName
                DC.L    HexType
                DC.L    Hex

                DC.L    EvenName
                DC.L    EvenType
                DC.L    Even

CapitalName:    DC.B    'Capital',0
HexName:        DC.B    'Hex$',0
EvenName:       DC.B    'Even',0

                SECTION DemoBlock,BSS
                CNOP    0,2

ComalStruc:     DS.L    1

                END
```

This source has to be compiled and linked. Public domain assemblers and linkers will do the work.

3 Programming modules in C.

It is a very simple task to write modules in C (provided you know how to program in C).

The interface part is written as (almost) normal Comal FUNC or PROC lines in a separate text file and then compiled by the compiler CompInterface (found in the ModuleDev directory on the Comal.Extras disk).

The initialization routine (optional), the signal routine (optional) and all other routines are written as normal C procedures or functions.

As an example let's make a module containing these three functions and procedures:

```
FUNC Even(i OF LONG) OF SHORT
FUNC Hex$(i OF ULONG)
PROC Capital(REF t$)
```

3.1 The interface part.

First let's make the interface part. The source text for the interface compiler CompInterface is written in an editor. The source looks like:

```
// Interface for Module Demo

  FUNC Even(i OF LONG) OF SHORT
  FUNC Hex$(i OF ULONG)
  PROC Capital(REF s$)
```

Note that comments and empty lines may be used as in Comal.

This source file is stored on disk as Demo.interface and then it is compiled by executing the following shell (CLI) command (provided Demo.interface and CompInterface are placed in the current directory)

```
CompInterface Demo.interface
```

The output from CompInterface is an assembler source text with the name Demo.interface.a. This is compiled with a normal assembler (for instance the SAS/C assembler) by executing the command (the include file Comal.i must be in the current directory):

```
lc:asm Demo.interface.a
```

The output from the assembler is an object file Demo.interface.o ready to be linked.

3.2 The procedures and functions.

The routines in a C programmed module are made as normal C procedures and functions. The names and the types are those used in the interface source. You only have to know that strings (marked with a dollar sign ($) in the interface source) must be changed to 'char *' and that the dollar sign are omitted from the name (see the function Hex$ in the example).

The C source for the module we are making as an example looks like this:

```
/* Module Demo */

#include <exec/types.h>
#include <string.h>
#include <ctype.h>
#include "Comal.h"
#include "Comal_protos.h"

short Even(LONG i)
{
  return( (i+1) & 0x01 );
}

char *Hex(ULONG i)
{
  char *String;

  /* Place return string in workspace    */
  String = ComalStruc->CurrWorkBottom;
  if ( ComalStruc->CurrWorkBottom+14 >= ComalStruc->CurrWorkTop )
    ErrorText("Out of memory");
  ComalStruc->CurrWorkBottom += 14;
  String[0] = '$';
  (void)stcl_h(String+1,i);
  return( String );
}

void Capital(char *Str)
{
  while ( *Str )
  {
    if ( islower(*Str) )
      *Str = *Str - ('a'-'A');
    Str++;
  }
}
```

This program is stored on disk under the name Demo.c and then compiled by executing the command (SAS/C compiler)

```
lc:LC -b0 -fi -oDemo.o Demo
```

The output from the compiler is an object file ready to be linked.

3.3 Linking the object files.

The object files made by the assembler and the C compiler has to be linked together. This is done by the following command (again the SAS/C linker is used):

```
lc:blink Demo.interface.o Demo.o TO Comal:Modules/Demo
                LIBRARY Comal.lib lib:lcnb.lib lib:Amiga.lib
```

(type in as one line).

The output from the linker is a module Demo placed in the directory Comal:Modules and the module can be used by placing the line

```
USE Demo
```

in a Comal program.

3.4 An initialization routine.

The interface compiler CompInterface makes the basic initialization code for you. If you need to do more initialization you have to place the following function in your code (NOTE! The name must be ModuleInit):

```
short ModuleInit(void)
{

    :

}
```

The function returns zero if the initialization succeded. Otherwise an error code greater than 5 is returned.

Example:
Here is a simple initialization routine:

```
short ModuleInit(void)
{
  if ( !(DOSBase = OpenLibrary("dos.library",0)) )
    return(100);
  else
    return(0);
}
```

3.5 A signal routine.

If you need to receive signals from the Comal system a signal procedure has to placed in your code (NOTE! The name must be signal):

```
void signal(short s)
{
  :
}
```

Example: A signal routine is very often used to close libbraries.

```
void signal(short s)
{
  switch(s)
  {
    case SIG_CLOSE:
    case SIG_DISCARD:
      if( DOSBase )
      {
        CloseLibrary(DOSBase);
        DOSBase = NULL;
      }
    case SIG_CLEAR:
    case SIG_RUN:
    case SIG_STOP:
    case SIG_END:
      break;
  }
}
```

3.4 Using the script file MakeMod.

All the command used to make a module is placed in a script file MakeMod (found in the ModuleDev directory on the Comal.Extras disk). Provided that both source files (Demo.interface and Demo.c) are placed in the same directory as MakeMod and Comal.h, Comal_protos.h, Comal.i, and Comal.lib the module can be made simply by executing the command:

```
MakeMod Demo
```

3.5 The interface compiler CompInterface.

The interface compiler CompInterface takes as input a text file containing normal Comal procedure and function heads. The command template is:

```
CompInterface [options] filename
```

Options are specified as a hyphen followed by a single letter. Current options are:

-o This option is followed by a directory path. The output from the compiler is placed in that directory. Default output path is the current directory.

-i If this option is used, the interface source is added as the modules informationtext (the text you get when selecting the Show Modules menu item in the Program menu).

-p If this option is used, all functions (except string functions) can beused as procedures, too.

The output from CompInterface is an assembler source text that has to be compiled by a normal assembler.

4 Calling internal Comal routines from modules.

A number of routines in the Comal system may be called from a module. These routines are:

```
void ErrorNumber(short ErrorCode);
        -6               D2
```

Stop execution with error message. No return from the routine.

```
void ErrorText(char *ErrorText);
        -12            D0
```

Stop execution with error text. No return from the routine.

```
void ExecBreak(void);
        -18
```

Call this routine if the break flag is set in the Comal structure (see the include files). No return from the routine unless the execution are continued or the break is disabled (SET ESC statement).

```
struct Window *LockComalWindow(void);
        -24
```

Get a shared lock for the standard IO window. The routine will open the standard IO window (if not already opened) and return a pointer to the window structure.

```
void UnlockComalWindow(void);
        -30
```

Release the lock for the standard IO window.

```
void AddComalDevice(struct IoDevice *Device);
        -36                         A0
```

Add an IO device. The parameter is a pointer to an initialized IO device structure. This structure will be put into the start of the list of standard IO devices (like "ds:").

```
void RemComalDevice(struct IoDevice *Device);
        -42                         A0
```

Remove an IO device from the list of IO devices. The parameter is a pointer to the IO device to be removed.

```
unsigned long ComalWait(unsigned long SignalMask);
        -48                               D0
```

Wait for the signals in the signalmask and all the Comal signals. The routine ORs all the Comal signals to SignalMask and then calls the Exec routine Wait.

```
void AddExcept(struct ExceptStruc *Except);
        -54                           A0
```

Add an exception routine to the list of exception routines in the Comal system. The parameter is an initialized structure of the form:

```
struct ExceptStruc
{
  struct ExceptStruc *Next;
  ULONG SignalMask;
  void  (*ExceptRoutine)(ULONG,struct ExceptStruc *);
  APTR  IdField;
};
```

The field SignalMask is a mask containing the exception signals. ExceptRoutine is the address of your exception routine. This routine is called with a mask containing the signals that caused the exception and a pointer to the exception structure. The field IdField is for your own purpose.

```
void RemExcept(struct ExceptStruc *Except);
        -60                         A0
```

Remove an exception routine from the list of exception routines in the Comal system. The parameter is a pointer to the exception structure to be removed from the exception list.

```
void AddSignal(unsigned long SignalMask);
        -66                     D0
```

Add a signal to the Comal signals.

```
void RemSignal(unsigned long SignalMask);
        -72                     D0
```

Remove a signal from the Comal signals.

```
short GetAccept(char *Str, char *Accept, char *Cancel);
        -78           A0          A1            A2
```

Set up a requester. The parameter Str is the text written in the requester. This string may contain the C formatting character '\r'. The parameters Accept and Cancel are the text filled in the positive and negative gadgets respectively.

The routine returns TRUE if the positive gadget was pressed. Otherwise it returns FALSE.

The routines are organized as an Amiga library with jump vectors at negative offsets (the numbers shown under the names) from the Comal structure. From an assembler program it must be called with the address of this

structure in register A6. Here is a typical calling sequence:

```
        MOVE.L  ComalStruc,A6
        JSR     -LVORemSignal(A6)
```

All the routines uses the registers D0-D1/A0-A1. All other registers are unchanged.

5 Calling comal programmed routines from a module.

A Comal programmed procedure or function is called as a C procedure, i.e. the parameters are transfered on the stack and a return value is returned in register D0 (D0 and D1 if float).

Example:
We will make a module containing an integral function of the form:

```
    FUNC integral(f OF FltFnc, a, b)

    TYPE FltFnc=FUNC(FLOAT) OF FLOAT
```

The interface looks like:

```
// Interface for integral module

  FUNC integral(f OF FUNC OF FLOAT,a OF FLOAT,b OF FLOAT) OF FLOAT
```

and the code like this:

```
/* Integral module    version 92.07.29  */

#include <math.h>
#include "Comal.h"
#include "Comal_protos.h"

double integral(double (*f)(),double a, double b, struct ProcType *Inf)
{
  double xi, s, dx;
  int n;

  if ( (Inf->NumPar != 1) ||
        (*((ULONG *)(Inf->Param)) != (0x0000FFFF & FLOAT_ID)) )
    ErrorText("Illegal function parameter");

  n = 16;
```

```
    dx = (b-a)/n;
    xi = a;
    s = (f(a)-f(b))*dx/6.0;
    while ( (n--) > 0 )
    {
      xi = xi+dx;
      s = s+(f(xi)+2*f(xi-dx/2))*dx/3;
    }
    return(s);
  }
```

One thing has to be noted. At the call to a Comal programmed routine the register A4 must point to the data used by the Comal system. At the call of your own routine the register A4 has the correct value and if you do not touch this register, there is no problem.

But what about C compiled code? How can you be sure that the register A4 is not used by the code made by your compiler?

As a general answer to that question we have to say, that you cannot be sure! Fortunately it seems as if SAS/C (version 5.10) do not use the register if it is compiled in "non base relative" mode (use the nb-libraries) and if you do not use the math library (the one containing the functions sin, cos, tan, ...). Replacement for most of the functions are placed in Comal.lib.

6 Making a library interface.

From time to time new libraries for the Amiga appears (a lot of them are public domain). It is a very simple task to make an interface for such a library.

As an example we will make an interface for the ReqTools.Library (found on Fish disk #575).

First we have to make an interface source file. This is best done if you have a C proto include file for the functions in the library. Such a file is supplied with the ReqTools.Library. A small part of this file looks like:

```
  APTR rtAllocRequestA(ULONG, struct TagItem *);
  void rtFreeRequest(APTR);
  void rtFreeReqBuffer(APTR);
  LONG rtChangeReqAttrA(APTR, struct TagItem *);
         :
```

The proto file is changed to an interface file in this way:

```
// Interface for reqtools.library

  FUNC rtAllocRequestA type OF ULONG, TagItem OF ULONG) OF ULONG
  PROC rtFreeRequest(req OF ULONG)
  PROC rtFreeReqBuffer(req OF ULONG)
  FUNC rtChangeReqAttrA(req OF ULONG, TagItem OF ULONG) OF LONG
         :
```

Note that pointers are changed to ULONG.

Now we are going to make the code. This contains only an initialization and a signal routine:

```
/*********************************************************************/
/*                                                                   */
/*    ReqTools library - version 92.05.11                            */
/*                                                                   */
/*********************************************************************/

#include <exec/types.h>
#include <proto/exec.h>
#include <pragmas/exec.h>
#include "Comal.h"
#include "Comal_protos.h"

struct Library *ReqToolsBase = NULL;

short ModuleInit(void)
{
  if ( !(ReqToolsBase = OpenLibrary("reqtools.library",37)) )
    return(0);
  else
    return(150);
}

void signal(short s)             /* Signal routine */
{
  switch(s)
  {
    case SIG_CLOSE:
    case SIG_DISCARD:
      if( ReqToolsBase )
        CloseLibrary(ReqToolsBase);
  }
}
```

The files are compiled as usual. To link it you have to specify the ReqTools linker library:

```
  blink ReqToolsLibrary.Interface.o ReqToolsLibrary.o
    TO Comal:Modules/ReqToolsLibrary
    LIBRARY Comal.lib reqtoolsnb.lib Lib:lcnb.lib Lib:Amiga.lib
```

(NOTE! one line only).

X. APPENDIX

A. Customizing the Comal text.

All texts used in the Comal system are stored in two files Comal.texts and Comal.interpreter.texts found in the Prefs/Texts drawer. These files are simple ASCII files and it is a simple matter to edit these files.

The format of the files is shown in the following fraction of the file Comal.texts:

```
/* Other window title texts */
#052  "Comal - project"
#053  "Command - project"
#054  "Error - project"
#055  "Comal module library"
#056  "Comal module information text"
#057  "Comal watch window"

/* About window texts */
#058  "About Comal ..."
#059  "Comal version 3.01"
#060  "Total memory available:"
#061  " bytes of memory for program text"
#062  " bytes of workspace memory for running program"
#063  "Comal was designed and developed by:"
#064  "Svend Daugaard Pedersen"


/* Menu texts  */
#065  "Project"                          /* Menu name    */
#066  "New"                              /* Item name    */
#067  "N"                                /* Short cut    */
#068  "Open..."                          /* Item name    */
#069  "O"                                /* Short cut    */
#070  "Save"                             /* etc.         */
#071  "S"
#072  "Save As..."
#073  "A"
#074  "Print"
#075  "P"
#076  "Open Command Window"
#077  "K"
#078  "About..."
#079  ""
#080  "Clear Program Buffer"
#081  ""
#082  "Quit Project"
#083  "Q"
```

Each text is placed in one line and this line starts with the number of the text. The text itself is placed in double quotes and consists of ASCII characters or the C style codes '\n' (new line), '\"' (double quote) or '\\' (for the character '\').

All texts must be defined and the numbers must be placed in increasing order. The first line has number zero. Empty lines and the use of C style comments are allowed.

Note that each menu item and subitem text consist of two texts - one for the text itself and one for the menu short cut. If the short cut text is the empty text (the text ""), no short cut will be available.

Having made a new set of text files they must be stored in the directory Prefs/Texts using file names of the forms:

```
Comal.<text_id>
```

and

```
Comal.Interpreter.<text_id>
```

Normally the <text_id> is the name of a language but it could be anything (but the same for both files!).

B. The format of the AREXX macro file.

At start Comal searches for a file Comal.macro containing macro definitions. The format of a macro file is shown in the following example:

```
#0 SaveBlock
#1 SaveModule
#2 SaveModule3
#3 StartProgram
#4 'StartProgram Comal:Programs/Integral'
```

Each macro line starts with a number sign (#), one digit (0,1,2,..,9), a blank (a space) and the macro string itself. If a line does not start with # it is treated as a comment.

The line starting with #0 is attached the function key F1, the line starting with #1 is attached the function key F2 etc.

C. Customizing Comal.lib.

The link library file Comal.lib found in the directory ModuleDev contains interfaceroutines for the Comal routines as well as initialization and termination code for a C programmed module.

The initialization and termination code is compiler dependent and the code found in Comal.lib is for use with the SAS/C compiler. In the directory ModuleDev you will also find a file Comal0.lib which is a library file with out the initialization and termination code. This one can be used to make a library file for other C compilers.

As part of the initialization a routine by name C_Init is called and when the module is removed a routine C_Term is called. These routines should do the initialialization and termination necessary for your C compiled code. To make it you should look at the source for the c.o startup code.

Here is the code used for SAS/C:

```
; Module startup for SAS C v.5.xx  -  version 92.07.29
;

               include  Comal.i

               XDEF     __base
               XDEF     _XCEXIT
               XDEF     __xcovf
               XDEF     _SysBase
               XDEF     C_Init
               XDEF     C_Term

               XREF     _LinkerDB
               XREF     ___fpinit
               XREF     ___fpterm
               XREF     _ComalStruc

AbsExecBase    EQU       $4

               SECTION  InterfaceCode,CODE
               CNOP     0,2

; Initialization routine
;
C_Init:        MOVE.L   _ComalStruc,A0       ; Get Comal structure
               MOVE.L   CS_MinStack(A0),D0   ; Get stack bottom
               ADD.L    #$200,D0             ; Calculate "safe" bound
               MOVE.L   D0,__base            ; .. and store
               MOVE.L   AbsExecBase,_SysBase ; Set Exec base
```

```
                MOVE.L  A4,-(A7)            ; Initialize floating point
                LEA     _LinkerDB,A4
                JSR     ___fpinit
                MOVE.L  (A7)+,A4

                CLR.W   D0                  ; Set status
                RTS                         ; .. and return

; Termination routine
;
C_Term:         MOVE.L  A4,-(A7)
                LEA     _LinkerDB,A4
                JSR      ___fpterm          ; Terminate floating point
                MOVE.L  (A7)+,A4
                RTS


; Report stack overflow (only used if stack check is turned on)
;
__xcovf:        MOVEQ     #5,D2
_XCEXIT:        MOVE.L    _ComalStruc,A0    ; Get Comal library base
                JSR       -6(A0)            ; .. and return

                SECTION   InterfaceBlock,BSS
                CNOP      0,2

_SysBase:       DS.L      1                 ; Exec library base
__base:         DS.L      1                 ; "Safe" low bound of stack

                END
```