



Composition

CompoScript Guide and Code Examples

ProAction

3QD
DEVELOPMENTS LTD

APDL

Composition

Copyright © Rob Davison, ProAction & 3QD Developments Ltd 2026
All rights reserved.

No part of this product may be reproduced in whole or part by any means without written permission of the publisher. Unauthorised hiring, renting, lending, public performance or broadcasting of this product or its parts is prohibited.

While every care is taken, the publisher cannot be held responsible for any errors in this product, or for loss of data or consequential effects from the use of this package.

Software and Documentation supplied by
APDL
a part of
3QD Developments Ltd
Single User version downloaded from the free of charge software section of
www.riscos.com

Please note that any free of charge downloads from RISCOS.com are supplied without any technical support package as per the licence agreement.

This document last edited on 29/09/26



If you have found this application and this manual useful please consider making a donation to help cover the costs of cataloguing, documenting, updating and hosting the APDL applications. Scanning the QR code will take you to our donations page.

An introduction to Composcript

What is Composcript? It is a method of automating a variety of tasks within Compo. It can be thought of as a macro language, which allows a whole series of actions within Compo to be collected together into a single file, which can if desired be activated by clicking on an icon in a toolbar. It can also be activated by dropping the script onto a Compo window.

Composcript can also be used to batch process files. It is possible to drop a directory onto a Composcript program and have every file in that directory processed and resaved. If you commonly perform the same operation on a large number of files, Composcript can save a very considerable amount of time.

You can write your Composcript in which ever editor you normally use, either Zap or StrongEd or even Edit will do.

Basics

Like most languages, Composcript uses variables, loops and procedures. If you are familiar with a modern programming language, you will find the ways Composcript handles these easy to learn.

Variables

There are two types of variable used by Composcript. You can declare variables yourself. These are either integers or strings. There are also a number of built in variables, which give information about objects. These built in variables can be recognised by the dot at the beginning of the object name.

```
Let fred = .canvaswidth / 4
```

Here, fred is a user defined variable and .canvaswidth is a built in variable (it gives the width of the current canvas).

Loops

There are a number of loops available.

The For Next and Repeat Until loops are unsurprising

```
let fred = 1
For i = 1 to 5
    fred= fred * i
next

repeat
    fred = fred + 1
until fred > 100000
```

Procedures

Procedures work on the currently selected object. They are defined in a similar manner to BASIC, but do not take parameters.

```
def procbloggs
let fred = fred +1
endproc

and are called by

let fred = 1
procbloggs
```

Selected objects

Many of the Composcript commands operate on the selected object. To select an object, you use the `select` command.

Examples are

```
select "Fred"
select CANVAS
select FIRST
select ALL
select NONE
```

These select the object called fred, the canvas, the first object, all objects and no objects respectively.

One frequently used feature involves a variable, a loop and the select command.

```
for loop = 1 to 5
  select "fred<loop>"
next
```

This selects the object fred1, fred2, fred3, fred4 and fred5.

There are several built in variables associated with the selected object. To examine these, you use a very useful command.

```
report "hello world"
```

The report command displays its argument in a window.

```
select "Fred"

report "object is <.fullname>"
report "height is <.height>"
report "Width is <.width>"
report "opacity is <.opacity>"
```

will display information about the object "Fred".

As will become apparent, you can do a great deal more with these variables than just display them.

Attributes

Some of the attributes of selected objects are more complex than height and width. A selected object may have a shadow for example. In this case, setting attributes can be done as follows.

```
select "Fred"
shadow ON
{colour:&a0a0a0 XOffset:3
YOffset:3
}
```

This sets various attributes of the shadow mask.
This is really just a shortcut for writing

```
.shadow.type = 0
.shadow.colour = &a0a0a0
.shadow.xoffset = 3
.shadow.yoffset = 3
```

Menus

Composcript allows you to provide menus for the user to select from. This is done via the `definemenu` command. Here is a complete Composcript example. Note that the first line of every Composcript file is

```
#CompoScript
```

Since Composcript files are just ordinary text files, there needs to be a way of recognising a Composcript file. The above line is how this is done.

Example Program 1

Here then is a complete Composcript program.

```
#CompoScript
# test menus

definemenu Shadows
{soft Superdooper-soft
Rather hard really
}

openmenu Shadows to result

if result <> "" then
report "I think the user selected item <result>"
else
report "Oops! the user didn't select any item!"
endif

end
```

Let's look at this program line by line.

```
#CompoScript
```

This is the first line of every Composcript file.

```
# test menus
```

Lines beginning with # are comments.

```
definemenu Shadows
{soft Superdooper-soft
Rather hard really
}
```

These lines define a menu. Shadows is the name of the menu.

```
openmenu Shadows to result
```

The `openmenu` command opens the specified menu. The item selected will be placed in the variable called `result`.

When the user selects an item, you can examine the variable to see which item, if any, was selected.

```
if result <> "" then
report "I think the user selected item <result>"
else
report "Oops! the user didn't select any item!"
```

Composition

```
endif
```

The selected item is just displayed here, but you could take different actions for each option. Note that variable names are enclosed in <> when they could be confused with a string.

```
end
```

This indicates the end of the program.

Prompts

Similar to the menu command is the prompt command, This allows you to ask the user a question.

```
#CompoScript

prompt to result
{you really want to do this pal?
Button1:Proceed
Button2:Crazy!
Button3:NoWay
}

report "Result is <result>"

end
```

Annotated examples

This section will introduce some additional features, by means of examples. Many of these examples are provided with Compo, so you can try them out for yourself.

```
#CompoScript
```

The standard header line.

```
let myname = "Rob Davidson"
```

create a variable containing the name of the illustrious author of Compo.

```
maketext "Titletext"
{is <myname>"
  Font:Homerton.Bold
  At: 128 128
  Size:48
  Angle:0
  Colour:&FF00
  Border:16x16
}
```

This creates a text object, specifying the font, position, size, rotation, colour and border.

```
select "Titletext"
```

Select the newly created object.

```
let .shadow = 1
```

Give the object a shadow.

```
makemask SHADOW copy BLEND
```

makes a mask. In this case a SHADOW mask, which initially is a copy of the blend mask.

```
processmask SHADOW Antialias Smudge AllButBlack
processmask SHADOW Edge Maximum
processmask SHADOW Antialias Smudge All
```

Processmask is a powerful command which allows you to perform actions on any of the masks. The parameters are the actions to take, which are those found in the mask editing section of Compo.

```
let .opacity = 127
```

This line sets the opacity of the selected object. The variable has a value between 0 and 256

```
let .shadow.colour = &707070
let .shadow.xoffset = 4
let .shadow.yoffset = -4
```

These lines set attributes of the shadow.

```
end
```

The standard end marker.

A more complex example.

Here is an example which generates an animation.

```
#CompoScript
```

The standard header line.

```
star "remove <.composcript$dir>.tempfile"
```

The star command passes its argument to RISC OS OSCLI command processor. In this case it remove a file called tempfile in the directory where the composcript file is located.

```
select "Text0"
```

Select the object called "Text0"

```
makemask tint copy blend
```

Creates a tint mask, which is initially a copy of the blend mask.

```
for xpos = -40 to .width+40 step 10
```

This starts a loop based on the object's width. The loop steps up by 10 each time round.

```
select "Text0"
```

Selects the object

```
PROC blended_animation
```

Calls a procedure

```
next
```

ends the loop.

```
for xpos = .width+40 to -40 step -10  
select "Text0"  
PROC blended_animation  
next
```

The same loop, but with xpos decreasing each time round.

```
star "wimptask intergif -loop -d 6 -best 255 -o <.composcript$dir>.anim/  
gif <.composcript$dir>.tempfile"
```

This line (it's should be all on one line) called the intergif utility to generate an animation from the files saved by this script. See later for how the files are created.

```
report "all done"
```

Report success.

```
select "Text0"  
makemask blend copy tint
```

Select the object and restore the mask to normal.

```
end
```

End of the main program.

```
defproc blended_animation
```

start the definition of a procedure.

```
nib up
```

stop the screen updating while the masking is in progress.

```
makemask blend copy tint
```

reset the mask to normal.

```
processmask BLEND Gradfill
{blended:yes Opacity:255
Centre: <xpos>,<.height>/2
Size:40
EdgeIntensity:0
CentreIntensity:255
}
```

Do a graduated fill in the mask. The fill is circular. The centre, size and opacity are specified. In this case, opacity has a value between 0 and 255. The effect of this is to make a small part of the text visible.

```
nib down
```

Allows the screen to be updated.

```
PROC export_snapshot
```

Calls a procedure.

```
endproc
```

End of proc blended animation.

```
defproc export_snapshot
definition of a procedure.

select CANVAS
export sprite
{mask:1 BlendCanvas:1
Append:YES
spritename:<loop>
}
```

select the whole canvas and exports the sprite as a file. The filename is specified. A mask is used. The sprite is appended to the existing file and the name of the sprite is the position in the loop. It is also possible to export as other types.

```
select first
```

Selects the object

```
endproc
```

Ends the procedure.

Loading/Saving dialogues

For batch processing, you need to find the directory the user wants you to process. You may also need to ask for a destination directory. This example does both.

```
#CompoScript
```

The standard header line

```
repeat
  dragload directory "Drag an input directory of images to process." to
  indir
until indir <> ""
```

Put up a dialogue and wait for a directory to be dropped onto it. When it is, the variable `indir` contains its name.

```
repeat
  dragsave directory "Output" "Drag the output directory to create." to
  outdir
until outdir <> ""
```

Put up a dialogue and wait for the user to drag the icon to disc. The default name in this case is Output. When the operation is complete, `outdir` contains the name of the directory.

```
fileinfo outdir
```

Fileinfo gets information about a file/directory

```
if .fileinfo.there = 0 then
  proc create_output_directory
endif

fileinfo outdir
if .fileinfo.there <> 2 then
  report "Output already exists (and is not a directory). Giving up."
  stop
endif
```

Check that it really has been created ok.

```
repeat
```

Start a repeat until loop.

```
scandirectory indir to found
```

Scandirectory reads the next available filename from a directory; in this case, from the directory `indir`, and places the result in the directory `'found'`.

```
if found <> "" then
  proc process_image
endif
```

If the last file has been read, `found` will be `""`, otherwise it will contain a filename. If it is a file, call a procedure.

```
until found = ""
```

Continue the loop until all files have been processed.

```
end
```

End of the main program.

```
defproc process_image
```

Start of procedure definition.

```
if .objects < 0 then
  proc make_canvas
endif
```

.objects holds the number of object on a canvas. If it is less than zero, there is no canvas. If that's the case, call a procedure to create one.

```
select all
delete
```

Delete any objects on the canvas.

```
loadimage found
```

load the image with the filename in the found variable.

```
select first
moveto centre centre
```

Select the object and centre it.

```
maketext "sometext"
{<.directoryscan.leafname>" Font:Homerton.Bold
  Size:25
  At:0 0
  Angle:0
  Colour:&FF0000
  Border:10x10
  Rubout:ON 30% &FF
}
```

create a text object called 'sometext'. The text is taken from the variable .directoryscan.leafname which holds the leafname of the last file read from a directory.

```
select "sometext"
moveto centre 0
```

Select the text and centre it in the x direction.

```
proc save_in_output_directory
```

Call a procedure to save it.

```
endproc
```

End of the procedure.

```
defproc save_in_output_directory
  select canvas
  export jpeg
  {quality:75 }
endproc
```

A procedure to export the canvas as a JPEG at 75% quality.

```
defproc create_output_directory
  star "cdir <outdir>"
endproc
```

A procedure to create a directory

```
defproc make_canvas
  makecanvas 768 576 0
endproc
```

Make a canvas.

Here are just a few of the other products you will be able to
download from RISCOS.com free of charge
(You can make a donation to help support future product releases but this is entirely voluntary)



This Documentation is © 3QD Developments Ltd 2026
Some of the products shown above might not be available immediately.
Not all products will run on all versions of RISC OS, check the product information page for details.

Other Products list last updated 29/2/26