



# **Confix**

## **Generic Configuration Utility**

### **Application Programming Interface**

© Copyright: 2002-2023, X-Ample Technology bv  
Author: Paul Reuvers  
Version 1.21 (01 May 2023)  
Status: Operational

### **Functional Specification**

## Disclaimer

---

Please note that the information given in this API is 'as is' and is subject to change without prior notice. Although every effort has been made to ensure that the information in this API is correct and complete, this cannot be guaranteed. Neither the author (Paul Reuvers) nor the company (X-Ample Technology) give any guarantee about the suitability of this information, or the software described herein, for any purpose or application, and can therefore not be held responsible for any damage, direct or indirect, arising from the use or misuse of this information and/or the software.

### **© Copyright 2002 - 2022**

Paul Reuvers  
X-Ample Technology  
Elzentlaan 43  
5611 LH Eindhoven  
The Netherlands

Phone: +31 (0)40 2940297  
E-mail: [support@xat.nl](mailto:support@xat.nl)

Latest release: <http://www.xat.nl/en/riscos/sw/confix/>

# Contents

---

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Using groups and icons	6
1.2	Different layouts	7
<b>2</b>	<b>Using !ConfiX</b>	<b>11</b>
2.1	Using the buttons	12
2.2	The !ConfiX menu	13
<b>3</b>	<b>Setting up your application to use !ConfiX</b>	<b>15</b>
<b>4</b>	<b>The Config file in more detail</b>	<b>17</b>
<b>5</b>	<b>The _Config file in more detail</b>	<b>19</b>
5.1	Header	20
5.2	Groups	28
5.3	Group header	28
5.4	Object descriptors	30
5.5	Greying out icons or groups of icons	35
<b>6</b>	<b>Data types</b>	<b>35</b>
6.1	Option	37
6.2	String	38
6.3	Integer	41
6.4	Floating point	42
6.5	Menus	43
6.5.1	Simple menus	43
6.5.2	Directory menus	44
6.5.3	Creating a filing system menu	46
6.5.4	TrueType font menu	47
6.5.5	External menu	48
6.5.6	Printer menu	49
6.6	Radio buttons	50
6.7	Font menu	51
6.8	Colour selector	52
6.9	Ruler	54
6.10	URL and e-mail	54
6.11	Buttons	55
6.12	Sprites	56
6.13	OLE	58
6.14	IP address and netmask	59
6.15	BlockDriver	60
6.16	Comment	62
6.17	Info	64
6.18	Slider	65
6.19	Spacer	xxx
6.20	Caption	xxx

<b>7</b>	<b>Calling !ConfiX from your application</b>	<b>69</b>
7.1	Passing parameters to !ConfiX	70
7.2	Messages from !ConfiX to your application	75
7.3	Messages from your application to !ConfiX	77
7.4	Loading the Config file	78
<b>8</b>	<b>Internationalising your software</b>	<b>80</b>
8.1	Using your application's Messages file	80
8.2	Using different _Config files	81

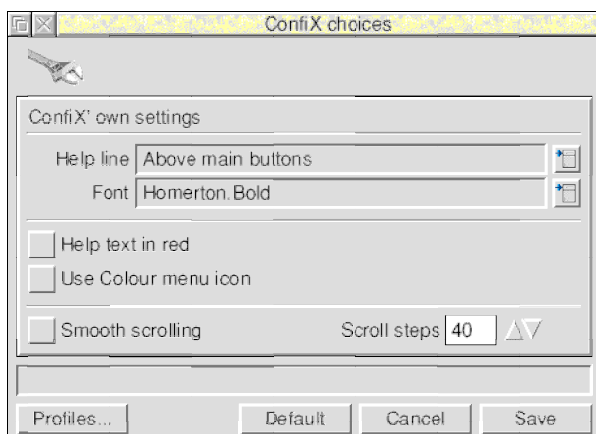
# 1 Introduction

Most RISC OS applications have a Choices file that allows the user to alter some of the default settings of the program. If you are writing your own applications, creating the Choices window(s) might be a tedious task, especially during the development of the program at which time requirements tend to change frequently. In most simple applications the Choices window is therefore omitted and the user is left to change the Choices file manually. In professional applications, a lot of the programmer's time is spent creating the Choices window(s) and the related code to enter and check the user's input.

!ConfiX is an attempt to create a generic utility that takes the pain out of creating Choices windows for each and every application and will produce a clear *Config* file that can be read and interpreted by your application easily.

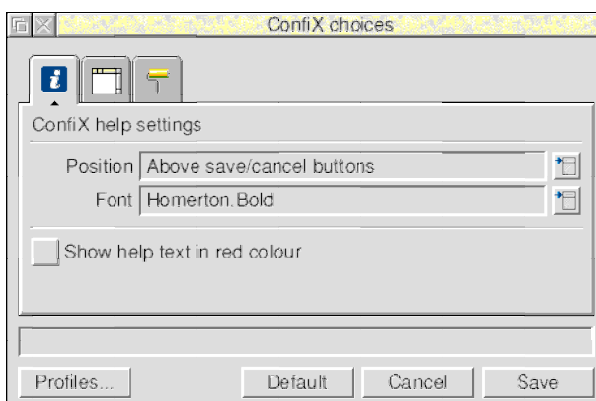
!ConfiX specifies a 'standard' choices file format that can be produced, altered and read by !ConfiX. Even a technically minded user, should still be allowed to alter the contents of the choices file manually. All a developer has to do is to supply a descriptor file (called *\_Config*) and provide minimum support in the application itself. The !ConfiX standard includes a whole range of controls, including numerical fields, strings, radio buttons, menu's, font selector, colour selector, etc.

!ConfiX uses the *\_Config* file, supplied by your application, to create its configuration window(s) on the fly and present them in a uniform manner. Once the windows are built, it loads the default choices file from your application (generally called 'Config') and opens the main configuration window. The user may then alter the settings and, when complete, click the *Save* button in the lower right corner of the window to confirm. !ConfiX will then issue a message to the application to indicate the fact the contents of the *Config* file have changed. Your application may then want to reload the new settings and act upon it.



For simple programs a single configuration window will be sufficient in most cases. The example on the left shows a single configuration window like the one used in previous versions of !ConfiX. A useful sprite is added to the top left.

If the *\_Config* file contains only a single group, !ConfiX will automatically create a window for you similar to this one.



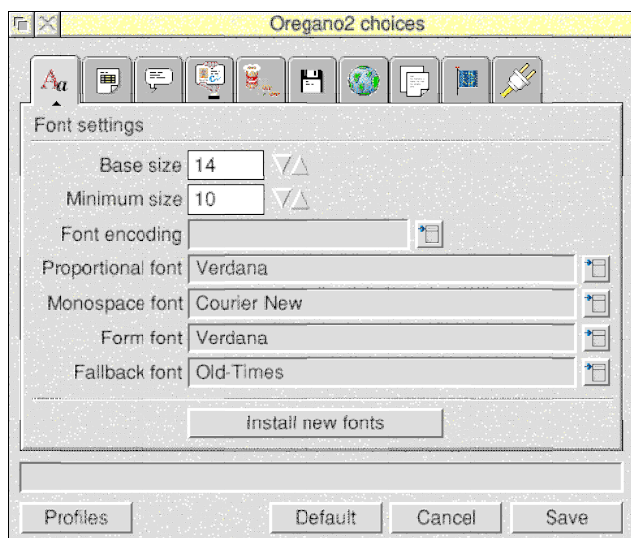
If the *\_Config* file contains multiple groups of settings, !ConfiX will automatically create a set of TABs for you, like in this window.

!ConfiX can of course also be used to alter its own settings. Open the menu (click **menu** whilst the pointer is inside the window) and select *Options...* A window, titled *ConfiX choices* will open, which looks like the one on the left.

Just above the buttons is the (optional) Help-line that can be used to present a more detailed description of the option currently under the mouse pointer. By default the Help-line is shown in a bold typeface so that it will attract the user's attention. You may however change the typeface and the position of the Help-line, or turn it off altogether.

## 1.1 Using groups and icons

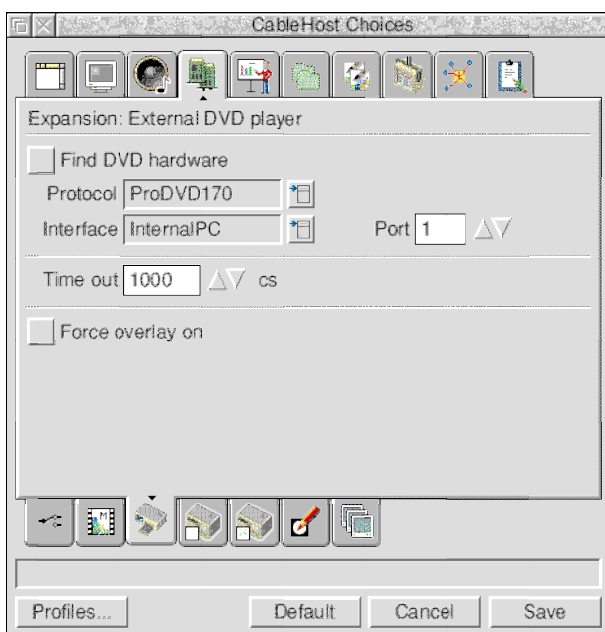
Complex programs may require a whole range of settings, which won't fit inside a single window. In such cases the choices may be divided into groups, each of which is identified by TAB, radio-button, icons, menu-entry, etc. For a typical application the choices window might look like this:



In this case, the program Oregano2 has 10 different configuration windows, each identified by a TAB. Rather than using names for the different TABs, !ConfiX uses icons, as these won't have to be translated for different languages.

The size of the window is under control of the `_Config` file supplied by the application.

As icons are used for the TABs, rather than textual names, more space is available to place the TABs. Very complex programs, however, may need even more TABs and the existing space may not be sufficient. Under such circumstances you might consider to increase the window width (which can be done easily), or create sub-groups by adding a second row of TABs at the bottom of the window, as the following example shows:



This example shows a program that requires several hardware extensions to be configured. The upper row of TABs is used to select the primary choices, *Expansion* in this case (i.e. the 4th TAB).

Once this TAB is selected, an extra row of TABs will appear at the bottom of the window. An extra TAB is now available for each hardware extension known by the program. In the example, the 3rd TAB is selected, which shows the settings for a DVD player.

The extra row of TABs is inserted automatically when sub-groups are defined in the `_Config` file (see chapter 5 for more information).

!ConfiX can be installed anywhere on your harddisc, but it is best placed inside the **!Boot.Resources** folder of your main harddisc, to ensure that it is 'seen' by the system before it is called by a client application.

Although a certain amount of programming is needed to make full use of !ConfiX facilities, it will be far less than creating the choices windows yourself for each and every application. Chapter 3 describes the structures and files needed in your application's resources to make best use of !ConfiX. Chapter 7 provides some detailed descriptions of the code needed to communicate with !ConfiX and some BASIC examples are given to get you going. Finally, chapter 8 has been added to convince you of the need to internationalise your applications.

From version 0.54 onwards, !ConfiX supports a number of different layouts, which enables the user to tailor ConfiX to his or her preferences. When creating `_Config` files for your applications, please check the configuration window regularly with the various layouts, to ensure a consistent layout.

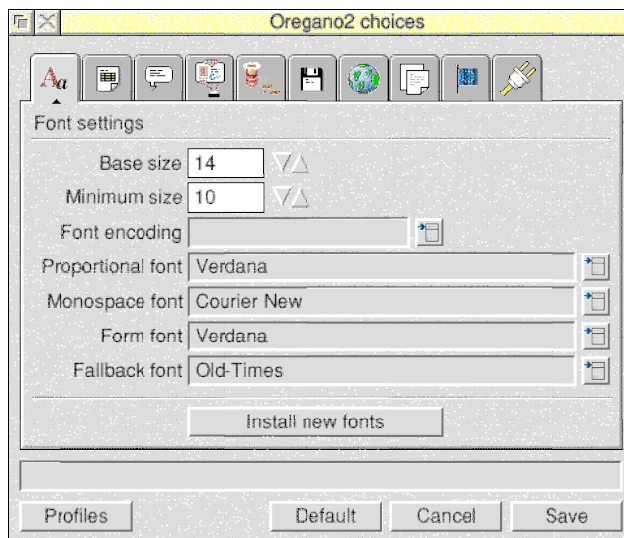
## 1.2 Different layouts

As some users prefer to use radio buttons rather than the more graphical TABs as shown on the previous page, ConfiX allows the user to select an alternative layout. This feature is present from version 0.54 onwards. The following examples show the same configuration, but each time in a different layout.

0

### Graphical TABs (default layout)

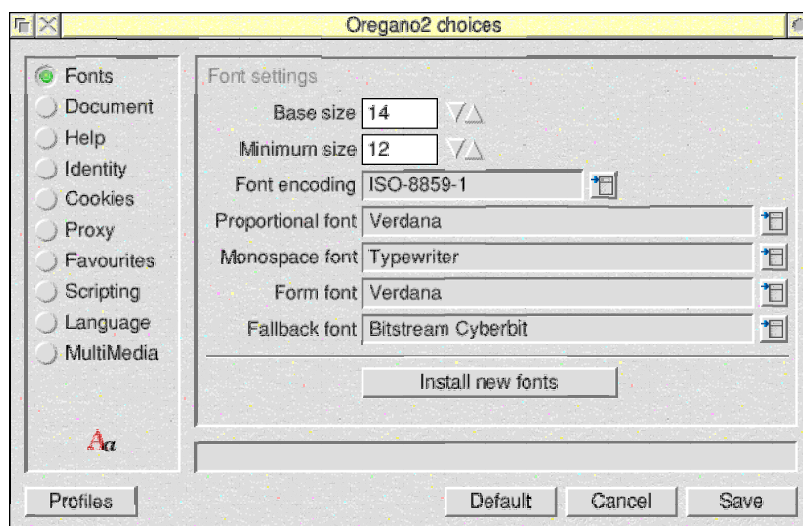
This is the default ConfiX layout. It uses a row of TABs at the top with icons in them, to represent the various configuration windows. Graphical TABs are very useful when internationalising your application, as they are language-independent.



1

### Radio buttons at left

This layout is used by many popular applications such as !StrongEd. A set of radio buttons, to the left of the window, can be used to select the various configuration windows.

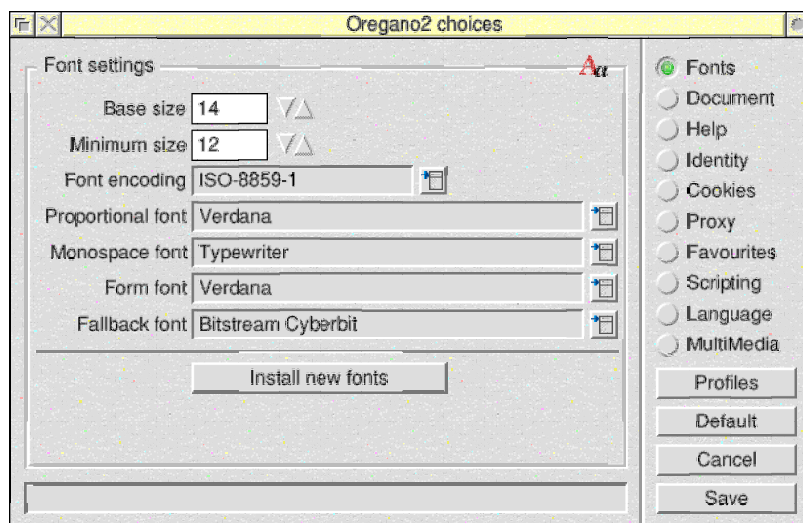


2

### Radio buttons at right

This is another popular layout used by many commercial and freeware applications, such as NewsHound and PopStar. A set of radio buttons to the right of the window is used to select the various configuration windows.

The main buttons will be placed just below the radio buttons.

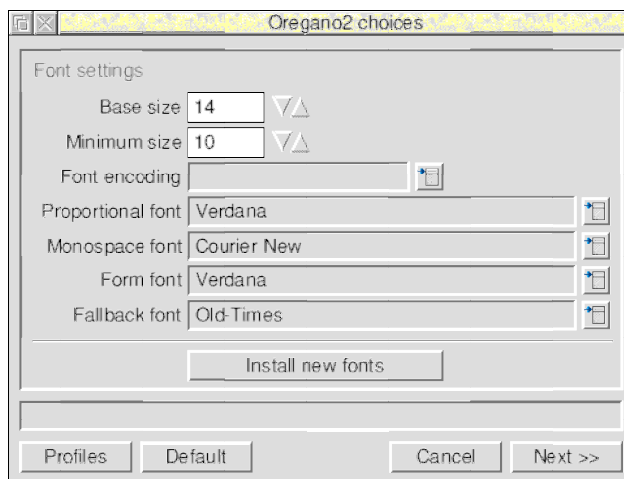


3

### Next / Previous buttons

If you don't want to use TABs at all, this may be the best choice for you. Instead of TAB icons, the buttons at the bottom are used to navigate through the various windows.

Click *Next* until you've reached the last window. The button then changes into *Finish*.  
Clickin this will save the settings.

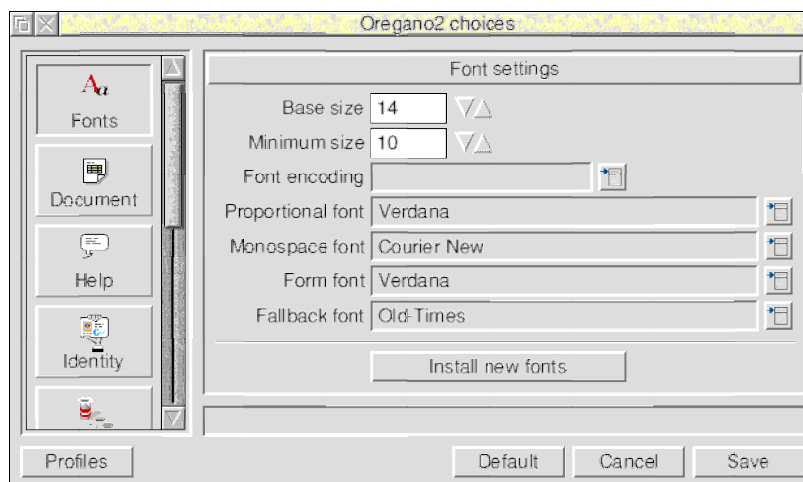


4

### Icon & Text buttons

This layout is similar to the one with the radio buttons on the left. However, rather than using radio buttons, a scrollable list of icons with the *Short* name underneath them will be present.

This layout is quite similar to the one used in Ovation Pro.



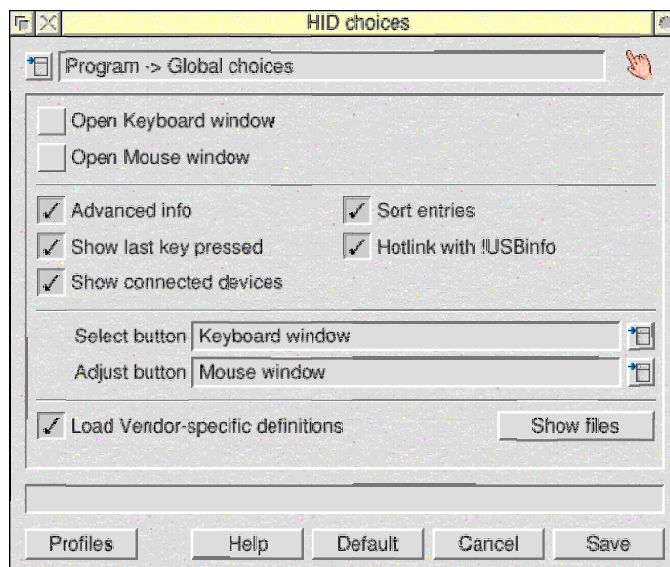
5

### Menu

This layout doesn't use tabs, icons or radio buttons to select a group. Instead, the group can be selected from a menu at the top of the window. The menu-icon can be placed in front of the window title (as in the example), or to the right of it.

Subgroups will be presented in sub-menus.

*This layout type has been added in version 0.90.*



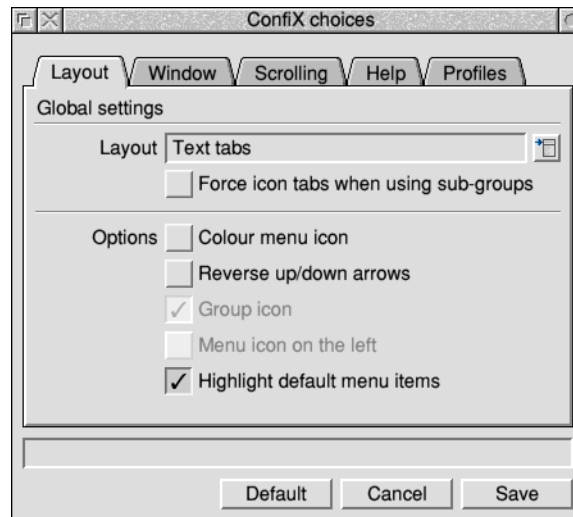


## 6

### Text tabs

This layout uses text-based tabs, all aligned at the top of the window. It is not suitable for displaying sub-groups..

*This layout type has been added in version 1.54 and is currently in an experimental stage.*



### Which layout to select?

It's difficult to predict which layout should be used for an application. Whenever possible, an application should not force a layout, but instead leave it to the user to select his or her preferred layout. As an application writer you should ensure that your `_Config` definition looks good in any of the available layouts.

### Default layout

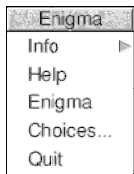
The first layout (Graphical TABs) is the only layout capable of showing sub-groups correctly. Therefore the Graphical TABs layout will be forced whenever sub-groups are used. If you don't want this behaviour, you need to turn off the option *Force default layout when using sub-groups* in ConfiX' own settings. This option is only available from version 0.77 onwards. ➤ See also paragraph 2.2.

### Help button

A *Help* button may appear at the bottom of the window. This button will only be present if that particular group contains a `Help` variable. Help buttons are only available from version 0.77 onwards.

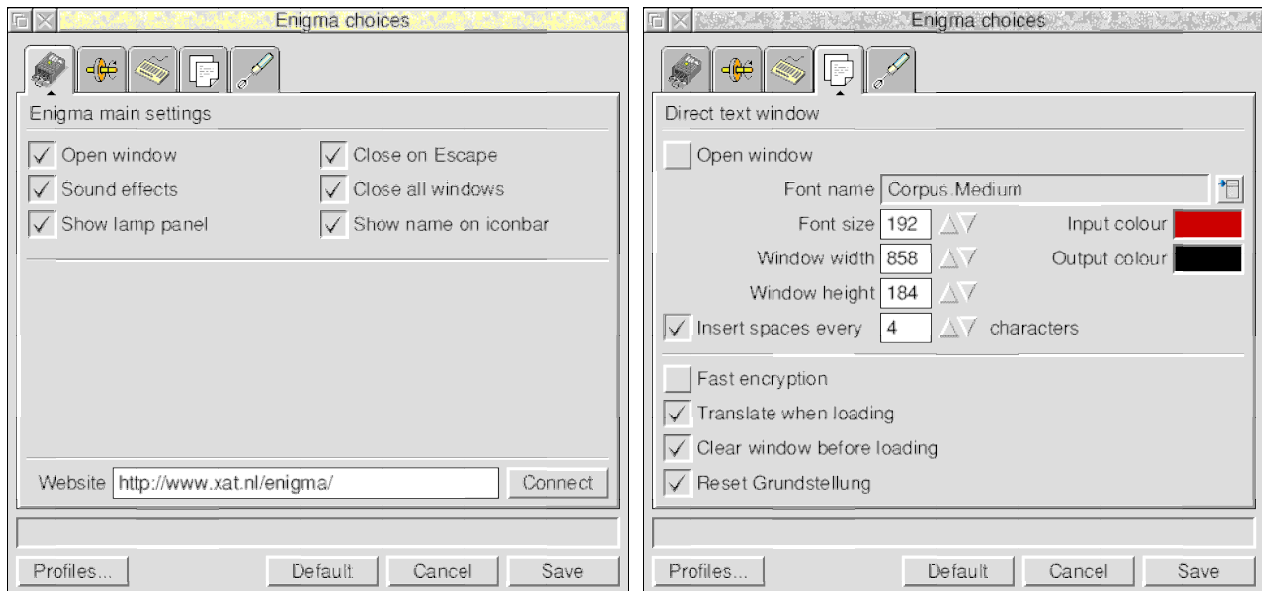


## 2 Using !ConfiX



Before we start using !ConfiX from our own application, we will first spend a little time examining its features. A typical application that uses !ConfiX is !Enigma, a freeware program that simulates a WWII encryption device. Like most applications, !Enigma allows the settings to be altered from the iconbar menu:

Once the user selects *Choices...* the application will launch !ConfiX and something like this will open:

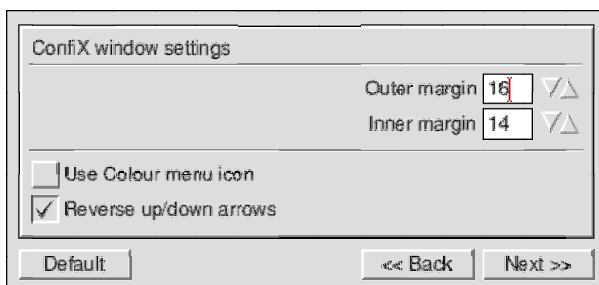


The application in this example uses 5 configuration windows, each represented by a TAB at the top of the window. Each TAB shows a useful sprite, supplied by the client application, and the title of the currently selected TAB is shown immediately underneath it. In the example, the currently selected TAB is called 'Enigma main settings'.

Whenever the user moves the mouse pointer over one of the other TABs (without selecting it), the Help-line (just above the buttons at the bottom of the window) will show the name of that particular TAB. Furthermore, whenever the mouse pointer is moved over one of the editable options inside the window, the Help-line will provide additional information.

Selecting another TAB reveals the related settings as can be seen in the rightmost picture above. Simply click the required TAB or press the TAB key on the keyboard. In a similar vein, Shift-TAB can be used to select the previous TAB, whilst Control-TAB will select the leftmost one. Of course the keyboard can only be used if the window has the input focus.

Depending on the client application, the ConfiX window may be completely different than in the above example. E.g. the **Cancel** and **Save** buttons at the bottom of the window, may be replaced by **Back** and **Next** buttons, or may be missing completely. If the window is configured to have **Back** and **Next** buttons, the TABs at the top of the window will be left out automatically. An example:



## 2.1 Using the buttons

---

The bottom of the window shows a fixed set of buttons that allows control of the settings. This paragraph describes the buttons in more detail. Please note that, depending the configuration of the window, you'll only see a subset of these buttons at any one time.

### **Save**

Once the user is satisfied with the new settings, they should be confirmed by clicking the **Save** button, or pressing **F3**. !ConfiX will then store an updated Config file inside your application, send a message to your application to indicate the the *Config* file has been changed, and then quit itself.

### **Cancel**

Whenever the user clicks the **Cancel** button, or presses the **Escape** key, no changes will be stored and !ConfiX will close down itself. However, before doing so, it will first issue a message to the application, to allow it to regain the caret (and hence regain the input focus).

### **Default**

In some situations, you might want to restore the initials settings of the application. Click the **Default** button to restore all configuration windows to their default settings (as defined by the application). You may now want to save the new settings by clicking *Save*.

### **Profiles**

To allow multiple configurations to be saved, e.g. for different situations or different users, !ConfiX uses a system of Profiles. Profiles can be saved from the !Config menu (explained later) and may be loaded by clicking the **Profiles** button in the bottom left corner of the window. If any profiles have been stored previously, a menu will open showing the various files. If no profiles are available, an error message will be displayed.

Rather than using a general button for this, ConfiX can be instructed to display a menu-icon instead, or add the profiles to the main menu (rather than displaying a button). If the use of Profiles is not desired, it may be turned *off* altogether.

### **Help**

From version 0.77 onwards, a *Help* button may be added automatically providing additional information on the current group. Whenever the *Help*-variable is present for a group, the *Help* button will be added automatically to the window. Selecting another group, for which no additional help is available, makes the button disappear again.

### **OK**

Optionally, ConfiX can display an OK button (in addition to the Save and Cancel buttons). It can be used for temporary settings (preferences) that are not as permanent as saved settings. Details about the use of the OK-button are available in chapter 7.1.

### **Next**

If this button is present, the ConfiX window will have no TABs. Clicking the *Next* button will bring you to the next TAB (i.e. the next configuration window). Alternatively you may press the *Tab*-key on the keyboard. Whenever you've reached the last TAB, the text in the button will change to *Finish* and clicking it will save the choices to disc.

### **Back**

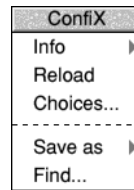
Use this button to go back on configuration window. Alternatively, you may press *Shift-Tab* on the keyboard. On the first TAB, the text in the button will change to Cancel and clicking will close the window without saving the choices.

### **Radio buttons**

Depending on the selected layout, a set of radio buttons may be present either at the left or the right hand side of the window. Clicking a radio button opens the corresponding configuration window. As this functionality is the same as for the TABs in the default layout, we will use the term TABs throughout this manual.

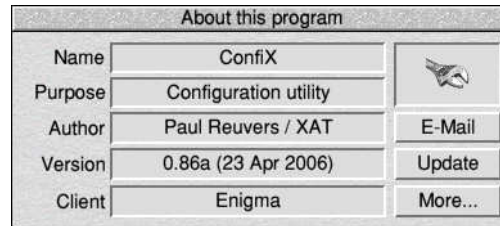
## 2.2 The !ConfiX menu

Some of the less obvious settings are hidden from the main window and can only be accessed from the !ConfiX menu. Just click **menu** whilst the mouse pointer is over the !ConfiX window. This will reveal a main menu similar to this one:



### **Info**

The first menu entry is **Info**, which leads to a common Info box, similar to the one shown here:



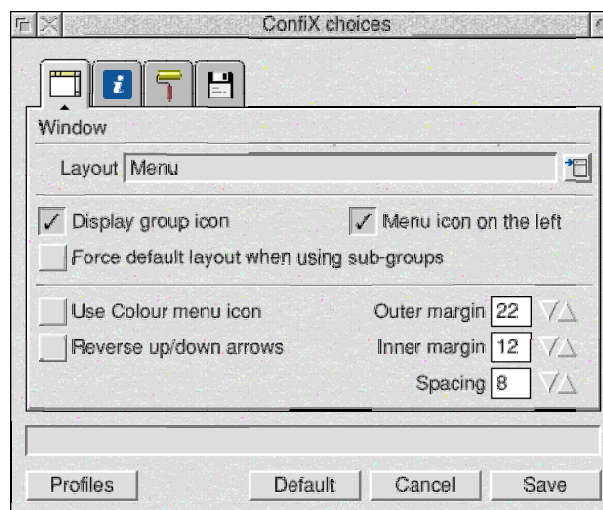
This window shows the current version of !ConfiX and provides buttons for easy access of the website where updates can be obtained. Click the **E-Mail** button to send an e-mail to the support department of XAT in case you experience trouble or have some useful suggestions that might help to improve the program further.

### **Reload**

Selecting **Reload** from the menu will reload the Config file again, which might be useful if you've altered some settings by mistake.

### **Choices...**

This entry allows you to alter the configuration of !ConfiX itself. When selected, a further !ConfiX window will open, allowing control of the appearance of !ConfiX and some of its default settings. It also allows the user to select the preferred layout.



The various settings are divided into groups, each of which can be edited by selecting the appropriate TAB. The first TAB allows you to set the main appearance of the window. The second TAB controls the use of the help system, whilst the third one allows you to turn on the (optional) smooth scrolling feature.

The last TAB gives access to some technical stuff that should only be changed if you know exactly what you are doing. It specifies which file systems will or will-not be seen in a *file system menu*.

### **Save as**

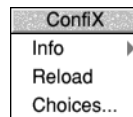
The **Save as** option can be used to store a Profile. It may be used to store the current settings as a named Profile for later use. Profiles will be stored in a directory called '\_Profiles' which resides at the same level as the 'Config' file (which will often be in '!Boot.Choices.AppName').

Profiles saved this way, can be reloaded by clicking the *Profiles* button in the configuration window and selecting the required profile from the menu. If no Profiles-button is present in the ConfiX window, any profiles will be added to the end of the main menu (i.e. *this* menu).

### **Find...**

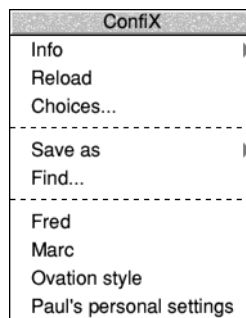
Selecting **Find...** will open the directory containing the profiles, so they can be edited or deleted manually.

The last two menu-items will only be available when the Profiles-system is turned on. If the Profiles-system has been disabled (by placing 'Profiles=0' in the header of the file '\_Config'), the main menu will look like this:



### **Profiles**

Rather than accessing the Profiles through the Profiles-button in the main window, an application can tell ConfiX to hide the Profiles-button and add any Profiles to the main menu instead. This is the case when 'Profiles=3'. The main menu will then look like this:

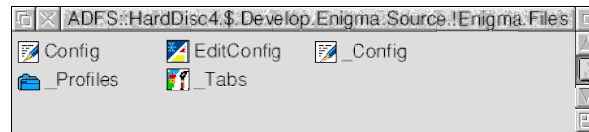


### **Notes about profiles:**

1. From version 0.85 onwards, the profiles directory (\_Profiles) is no longer stored inside the application, but will be created at the same level as your 'Config' file. This *can* be inside your application, but it can also be in a more useful place like, say, !Boot.Choices.YourApp.
2. From version 0.86 onwards, !ConfiX will accept long file names when saving a profile.

### 3 Setting up your application to use !ConfiX

If you want your application to call !ConfiX, you need to provide a minimum level of support, which will be described in chapter 7: *Calling !ConfiX from your application*. Additionally, your program needs to provide a set of files for !ConfiX to use. Although you are free to place these files anywhere inside your application, the following structure is suggested:



You may, of course, use the same *Files* directory to store additional information or configuration files. We will now examine each file in this directory briefly.

#### Config

This is the actual choices file that should be loaded by your application once it is started. Your application should also reload this file after receiving a WIMP Message 'Config' from !ConfiX. The exact file format of the *Config* file will be described later.

#### EditConfig

This file not required for correct operation of !ConfiX. It is however recommended, so that the configuration can be changed without the need to start your application. This may be useful, e.g. if your program was configured to use a dual serial card and, now the card is removed from your computer, the program refuses to start properly ending up in a recursive error. Double clicking the *EditConfig* file allows you to edit the *Config* file directly. The contents of the *EditConfig* file will be something like this:

```
Run <ConfiX$Dir> -res <MyApp$Dir>.Files -pos centre,centre -task MyApp
```

Please note that the **-task** switch in the above example contains the *taskname* rather than the *tasknumber* as described in the next chapters. As the application is not running, you won't be able to supply the task handle. As a last resort you'll then have to supply the task name instead. ➔ *See also chapter 7.*

#### \_Config

This file is rather crucial, as it describes the layout, syntax, help text, etc. of the !ConfiX window and the required output format for the Config file. Please note that the filename starts with an '\_' (underscore). A more detailed description of this file will be given later in one of the next chapters. When creating international versions of your software, you may want to use multiple copies of *\_Config*, one for each language. Please refer to chapter 8 for more information on this topic.

#### \_Profiles

This is an optional directory that can be used by !ConfiX to store any profiles. The user may load any previously stored profiles by clicking the *Profiles...* button in the configuration window.

#### \_Tabs

This is a Sprite-file containing a set of useful sprites to be used by !ConfiX. All of the icons needed to display the TABs should be supplied here. Additionally, you may include sprites that are used inside the configuration windows themselves. The !Enigma application from the example, contains the following sprites:



The icons should be created in a 256 colour mode, to ensure the best possible results in any screen mode. We also advise to make the sprite 30 x 30 pixels or smaller and use a mask whenever possible. We will see the above icon names again in the example files below.





## 4 The Config file in more detail

---

The structure of the *Config* file is rather simple. This is done for two reasons. First of all because it makes the implementation in your application easier as all variables can be read by a simple procedure. Secondly, it makes editing the file manually a lot easier. This might be necessary e.g. when you copy your program onto another computer without supplying the !ConfiX application.

The *Config* file is build from only two data types:

1. Groups
2. Variables

A group name is always enclosed in square brackets. It may optionally include the name of a sub-group, e.g.:

```
[FontSettings]
[Printing]
[Keyboard,Special]
```

Variables are always followed by an equal sign (=) and a value, e.g.:

```
FontName=Trinity.Bold
Size=100
Interval=350
```

Both group names and variable names should be treated case-insensitive to avoid confusion. Please note that the same variable names may be used in different groups. If such is the case, your application should treat them as different values.

A *Config* file can contain various variable types, such as strings, font names, integer variables, floating point variables, radio buttons etc. The type for each variable is defined in the *\_Config* file (described later). As an example we show part of the *Config* file from the !Enigma application.

```
| Enigma choices
| Saved Sun,20 Jan 2002.15:05:50 by !Config

Format=1

[Enigma]

AutoOpen=1
CloseOnEscape=1
Sound=1
CloseAllWindows=1
LampPanel=1
Name=1
WebSite=http://www.xat.nl/enigma/

[Rotors]

Realistic=1
Step=48
StepDelay=2

[Keyboard]

Space=X
Return=X
```

Lines starting with a '|' (vertical bar) should be ignored by your application as they are commenting lines. The entry 'Format=1' is optional and may be added if required by your application. If it is present, !ConfiX will verify this number against the value in the *\_Config* file before continuing..



## 5 The `_Config` file in more detail

---

The most important file in the whole !ConfiX system is the `_Config` file. It describes the size of the window, the required TABs and all of the options inside the windows. Each icon inside the configuration window can be fully controlled. Input fields may be checked for any given limits and the size of any input fields can be fully controlled as well.

The `_Config` file consists of a number of structures:

1. Header
2. Groups
3. Group Header
4. Object descriptors

Additionally commenting lines may be added by preceding them with a `'|'` (vertical bar). The example below shows the data for the first TAB of the !Enigma program.

```
| Enigma Configure definitions

Title = Enigma choices
Column = 450
LineUp = 0
Debug = 0
Width = 942
Height = 584

[Enigma]

Icon = enig
Text = Enigma main settings
Short = General
Width = 132

{
  Name = AutoOpen
  Text = Open window
  Type = Option
  Default = on
  Help = Open the main window when the program is started
};
{
  Name = CloseOnEscape
  Text = Close on Escape
  Type = Option
  Default = on
  Help = Close the main window when the Escape-key is pressed
}
{
  Name = Sound
  Text = Sound effects
  Type = Option
  Default = on
  Help = Sound effects when a key is pressed or a wheel is moved
};
{
  Name = CloseAllWindows
  Text = Close all windows
  Type = Option
  Default = on
  Help = Close all windows when the main window is closed
}
{
  Name = LampPanel
  Text = Show lamp panel
  Type = Option
  Default = on
  Help = Open the lamp panel (if present) when opening the main window
};
{
  Name = Name
  Text = Show name on iconbar
  Type = Option
  Default = on
  Help = Show Simulation name under the iconbar-icon
}
{
  Name = Ruler
  Height = 20
}
```

## 5.1 Header

---

The file header consists of a number of variables, each followed by an '=' sign and a value. Additional spaces may be used for clarity. The first line in the header e.g. (Title=**Enigma choices**) is used to describe the name to be displayed in the window title bar. This section describes all of the variables that can be used in the header of the \_Config file. Where applicable, the default value will be given in '[' ']' square brackets. The following variables may be used in the header:

Title	Window title (application name)
Width	Window width in OS units
Height	Window height in OS units (or '0' for auto-height)
Column	Column width in OS units
File	Alternative configuration file name
Load	Specify path for loading config file
Save	Specify path for saving config file
FileType	Config file type
Format	File format version number
SysVar	Unique system variable prefix for holding field values
CheckVersion	Minimum required !ConfiX version
Separator	Separator between variable and value
Remark	Character to mark a line as 'comment'
UseGroups	Enable/disable the use of Groups
Boolean	Representation of 0/1 values
IgnoreEmpty	Empty variables are not written to config file
MainButtons	Main window button types
Tabs	Control appearance of TABs
Default	Enable/disable 'Default' button
Profiles	Enable/disable 'Profiles' button
Help	Position of the Help line
HelpButton	Enable/disable 'Help' button
UseWebColours	Select between RGB or web-based colours
ColourFormat	Colour format (RGB, HSV, etc.)
ColourAuto	Inherit colour format from Config file
ProfileButtons	Type of buttons below Profile window
ProfileWidth	Width of the profile list in OS units
DateFormat	Default format for presenting dates
Output	Format of the Config output file

These variables are explained in more detail in the following paragraphs.

### 5.1.1 Mandatory settings

The following settings are mandatory and should not be omitted from the header.

#### Title

This variable is followed by the text that should be displayed in the title bar of the !ConfiX window. This will generally show the name of your application, such as: 'MyApp choices'. In this case the line should read:

```
Title = MyApp choices
```

#### Width

This variable specifies the width of the configuration window in OS-units. If the variable is omitted a width of about 942 OS-units is used instead, but this value cannot be guaranteed as it may change in future versions of !ConfiX.

#### Height

This variable specifies the height of the configuration window in OS-units. If the variable is omitted, a height of about 584 OS-units will be used instead. A value of 0, or indeed any small value, can be used to automatically adjust the height of the window to the minimum size needed to fit the largest window. If you want the size to be adjusted automatically, 'Height=0' would be recommended.

#### Column (or ColumnWidth or ColWidth)

In case you want to display two columns of variables side by side, this variable specifies the width of a single column on OS-units. As a rule of thumb you may want to use about half the `Width` for this in most situations.

### 5.1.2 Optional settings

The following variables have been added to allow !ConfiX to be used for applications that, for one reason or another, cannot comply with the ConfiX specification. This may be the case for programs that already exist and for which a change in the source code is not expected.

#### File

In most cases, the choices file will be stored inside the *Files* directory of your application and the name will be *Config*. However, there may be situations where the choices file must be stored in a different location, outside the application. If such is the case, the filename may be supplied in the header, e.g.:

```
File = <Choices$Write>.MyApp.Settings
```

If the `File` variable is omitted, the program will look for a text file called 'Config' inside the same directory as the `_Config` file. It is advised that applications store their choices in the computer's !Boot structure, by using the system variable `<Choices$Write>`. If the `File` variable is supplied, it will be used both to read and write the Config file. If you want to specify an alternative path for reading and writing, please use the variables below.

#### Load

This variable can be used to specify *only* the path from which the Config file will be read, e.g.:

```
Load = Choices:MyApp.Settings
```

Please note that the specified path may be a multi-part path in this case. It allows ConfiX to load the initial Config file from a different location (e.g. the `ROxxxHook` inside !Boot), when the required Config file hasn't been saved in '!Boot.Choices' yet. This feature can be overridden by supplying the '-load' parameter in the command line when launching ConfiX (see chapter 7). *This variable has been added in version 0.86 of ConfiX.*

#### Save

This variable can be used to specify *only* the path to which the Config file will be written, e.g.:

```
Save = <Choices$Write>.MyApp.Settings
```

*This variable has been added in version 0.86 of ConfiX.*

## FileType [Text]

By default the filetype of the Config file generated by !ConfiX is 'text'. If another filetype is needed by your application, use the FileType variable to define it, e.g. `FileType=Data`.

## Format

In cases where your application needs to check the format of the current Config file, the optional 'Format' variable may be added to the file header. If it is specified, !ConfiX will check against the given number. If it is not equal, an error will be reported and !ConfiX will abort itself.

## SysVar

If present, ConfiX dynamically sets a system variable for each item and selection made by the user. This can be used, e.g. to make the contents of a menu depend on another setting or selection. For example, when adding this line to the file header:

```
SysVar = MyAppVal
```

a unique system variable would be created for each item, in the following format:

```
<MyAppVal$Value$Group@Variable>
```

in which:

<code>MyAppVal</code>	is the name of the system variable (select your own unique variable prefix here).
<code>\$Value\$</code>	is the part added by !ConfiX.
<code>Group</code>	is the name of the Group to which the object belongs.
<code>@</code>	is the separator between the Group and the Object (must be '@')
<code>Variable</code>	is the name of the Object.

The value of the system variable will change dynamically as and when the user makes a selection. When closing the ConfiX window, all related system variables will be cleared again. If your application is called 'myApp', don't use 'myApp' as the SysVar prefix, as they will be cleared when closing the ConfiX window. Instead use some extension to the name, e.g. 'myAppVal'.

## CheckVersion

This setting can be used to ensure a minimum version of !ConfiX that may be needed to allow certain features to work. If your current version of !ConfiX is lower than the specified version number, an error will be produced. E.g.:

```
CheckVersion = 0.83
```

means that the application needs at least version 0.83 of !ConfiX. If the setting is omitted, any version of !ConfiX will be accepted by the application, however, if your application relies on certain features that have been implemented in !ConfiX at some stage, it may be useful to ensure a minimum version number. The minimum version numbers for most features are given in this manual.

## Separator [=]

By default !ConfiX uses the equal sign (=) to separate variables and their values. Some software however, may need a different separator, e.g. a semi-colon (;). If such is the case, use this variable to define it.

## Remark [|]

By default, the vertical bar '|' is used to enter remarks in the output file. If another separator is needed, use this variable to specify it. Please note that some application may require commenting lines to be preceded for example by a # sign.

## UseGroups [1]

!ConfiX uses a system of groups and variables. Each group will be represented by a different TAB in the configuration window. If, for some reason, your application cannot support group items in the config file, you may turn them off altogether. By supplying the line `UseGroups = 0`, the system of Groups is turned off and no TABs will be used. If the variable `UseGroups` is set to 2, group items will be hidden by placing a 'comment' character before it (i.e. a '!' or '#' sign). This way, they are unseen by the application, but !ConfiX can still use them to identify the various TABs. The following values may be used:

- 0 No groups (and no TABs)
- 1 Use groups and TABs
- 2 Don't use groups, but do use TABs \*)

\*) This feature was broken in versions prior to 0.86.

## Boolean [0]

Boolean variables are used to implement on/off features, such as a check box. To turn a feature on or off, one would use something like:

```
Enable=1
Enable=0
Enable=false
```

The values '1' and '0' are used to set the variable. In addition to this, you may also use the alternative methods to set a boolean variable, e.g. on/off, true/false, yes/no. !ConfiX will recognise all these variants and will set the variable appropriately. When writing the variable back to the Config file, !ConfiX will, by default, use the '1/0' notation. If another format is needed, you may set the Boolean variable as follows:

- 0 1 / 0
- 1 Yes / No
- 2 True / False
- 3 On / Off
- 4 true / false all lower case variant

## IgnoreEmpty [0]

If this option is set, any undefined variables (i.e. variables with no value, e.g. 'Enable=') will not be written back to the Config file. This way, only the explicitly defined variables are stored, allowing the application to use sensible defaults for any missing variable.

## MainButtons [1]

This variable can be used to control the appearance of the main buttons in the configuration window. The following values can be used at present:

- 0 No buttons will be displayed.
- 1 Standard *Cancel* and *Save* buttons will appear at the bottom of the window.
- 2 *Back* and *Next* buttons will appear at the bottom of the window.

## Tabs [1]

This variable is used to control the appearance of the TABs in the window. Please note that a setting of `MainButtons=2` will force the TABs to be turned off. When the TABs are invisible, you need to press TAB or *Shift*-TAB to move through the various groups. The following values can be used:

- 0 Turn TABs off.
- 1 Show TABs at the top of the window.

### Default [1]

By default a *Default* button will be present in the configuration window. If it is not wanted, it may be turned off by specifying `Default=0`. Please note that this option can be overridden by specifying the `'-default'` switch in the command line when launching !ConfiX. The following values can be used:

- 0 Don't show *Default* button.
- 1 Show *Default* button at the bottom of the window.

### Profiles... [1]

By default a *Profiles* button will be present in the configuration window. If it is not wanted, it may be turned off by specifying `Profiles=0`. Please note that this option can be overridden by specifying the `'-profiles'` switch in the command line when launching !ConfiX. The following values can be used:

- 0 Don't show *Profiles* button
- 1 Show *Profiles* button in the button area of the window
- 2 Show *Menu*-icon <sup>1</sup>
- 3 Show profiles in the main (ConfiX) menu <sup>2</sup>
- 4 ConfiX behaves as a Profile Manager <sup>4</sup>
- 5 Button launches Profile Manager

#### Notes:

1. A value of 2 will show a *Menu*-icon, *only* when the buttons are displayed at the bottom of the window. In all other cases, a standard button will be displayed. E.g. layout 2 (RadioRight) will show a standard 'Profiles' button, rather than a Menu-icon. This is done in order to keep the window design consistent.
2. When using 'Profiles=3', no buttons or menu-icons will be displayed. Instead, the profiles are available from the main menu (click **Menu** inside the ConfiX window).
3. Selecting 'Profiles=0' turns the profiles mechanism off. From version 0.86 onwards, the 'Save as' and 'Find' options in the menu will also be hidden.
4. The Profile Manager can also be selected via other methods. See paragraph 7.1 for further details.

### Help

By default the position of the Help line is specified in the !ConfiX' own settings. It is recommended that you use this situation in preference, as it will allow the user to set the overall appearance of the application whilst maintaining consistency between applications. In some situations however, it might be necessary to alter the position of the Help line, or leave the Help line out altogether. if the Help variable is specified in the `_Config` header, it will override the default settings as configured by the user. The following values can be used:

- 0 Help line off.
- 1 Just above the main buttons at the bottom of the window.
- 2 At the bottom of the window (below the main buttons).
- 3 At the top of the window.

### HelpButton

This variable allows you to override the configured behaviour of the *Help*-button. When set to '1', a *Help*-button is added to the window automatically whenever additional help (for the current group) is available.

- 0 Don't use *Help*-button.
- 1 Allow *Help*-button to appear.

### UseWebColours [0]

By default, colours are stored as `<red>`, `<green>`, `<blue>`, e.g. red will be stored as **255,0,0**. When creating applications for internet usage however, e.g. a web browser, colours are generally stored in web-format, e.g. `'#FF0000'`. !ConfiX may be forced to use web colours by setting the variable `UseWebColours` to '1' (one).

#### Note:

This variable is now deprecated and is supported only for backwards compatibility. If you want to use webcolours, use `'ColourFormat=4'` instead (see below).



## GroupIcon

From version 0.90 onwards, a group icon can be displayed in layouts that would otherwise not show an icon. By default, this feature is **on**. The user can configure the behaviour of this feature in the standard ConfiX Choices, however, the variable `GroupIcon` allows you to override this setting.

- 0 Don't display the group icon
- 1 Display the group icon in layouts that otherwise do not show icons

In turn, this variable can be overridden by the command line option '`-groupicon <0|1>`' (see paragraph 7.1).

## ColourFormat [14]

By default, !ConfiX specifies a colour in *simple* RGB notation. The colour components, *red*, *green* and *blue*, are all represented by a decimal number in the range 0-255, separated by commas. Optionally, the transparency of a colour may be added to the end of the string [0-255]. E.g.:

255,0,0	<i>represents red</i>
255,255,255	<i>represents white</i>
0,0,0	<i>represents black</i>
100,255,100,128	<i>represents bright green with half transparency</i>

Although this colour format is widely used, it is rather limited and doesn't allow for future extensions. Instead, you may specify an alternative colour format to be used. This can be done at object level (using the `Format` variable), or in the '`_Config`' header (using the `ColourFormat` variable). When specified, the `Format` defined at object level takes preference over the one specified in the '`_Config`' header.

The default `ColourFormat` may be specified as a number or by its name. E.g.:

1	RISCOS	Use RISC OS notation, e.g.: &0421FF00
2	RGB	rgb(255,33,4)
3	RGB%	rgb(100%,30%,2%)
4	FullHex	#FF2104
<b>14</b>	<b>Simple</b>	<b>Use the (default) simple notation: 255,33,4</b>

For a full list of supported colour formats, please refer to the description of the `Format` variable in chapter 6.8. This variable was added in version 0.86 of !ConfiX.

## ColourAuto [0]

When setting this variable to '1', each colour will inherit its format from the current notation in the 'Config' file. This overrides the setting of the `ColourFormat` variable (above) and the `Format` variable in the object definition. This variable was added in version 0.86 of !ConfiX. The following settings of `ColourAuto` are allowed:

- 0 Don't use automatic colour formatting
- 1 Inherit the colour format from the notation of the colour in the 'Config' file.

## ProfileButtons [1]

When using ConfiX in Profile-mode, a scrollable list with the available profiles appears to the left of the ConfiX window. It allows multiple version (profiles) of the Config file to be saved, copied and loaded. Below the scrollable list are small buttons for adding, deleting, copying and locking profiles. The appearance of these buttons can be controlled by setting of `ProfileButtons`. The following values are possible:

- 0 Default (hard coded)
- 1 RISC OS style**
- 2 Inline with other buttons
- 3 Same as (2) but with horizontal spacing
- 4 Text-based buttons
- 5 Apple Style

## ProfileWidth [400]

This variable specifies the width of the Profiles list in OS units. The default value is 400, whilst the minimum width is 250 (this may change in a future version).

## DateFormat [0]

This variable specifies the default format for presenting dates. The following values are possible:

0	Default	DD MMM YYYY	31 Dec 2019
1	EU, EUR	DD-MM-YYYY	31-12-2019
2	US/, USA/	MM/DD/YYYY	12/31/2019
3	US-, USA-	MM-DD-YYYY	12-31-2019
4	DE, DL, BRD	DD.MM.YYYY	31.12.2019
8	ISO, XML	YYYY-MM-DD	2019-12-31
9	Numeric	YYYYMMDD	20191231

## Output [0]

This variable can be used to alter the format of the Config file. By default, the configuration is used in a standard human-readable text format. The following output formats are currently known:

0	Text	Standard text-based format	
1	TOML	Tom's Obvious Minimal Language	Added in version 1.52

### 5.1.3 Changing window furniture

In some situations it may be useful to control the appearance of the various window tools of the configuration window. E.g. it might be useful to leave the *Close* icon out if you want the user to change all settings before leaving the program. The following variables have been added to allow configuration of the window:

**CloseIcon [1]**

**BackIcon [1]**

**TitleBar [1]**

Changing any of these variables to 0 (zero), will turn the specified attribute off. Please note that if you turn off the TitleBar, all other attributes in the TitleBar will be turned off as well.

### 5.1.4 Changing the layout

#### Layout [0]

From version 0.54 onwards, ConfiX allows the user to select an alternative layout. When creating your own *\_Config* files, you need to ensure that your configuration windows appear correctly with each layout selected.

Alternatively, you may force ConfiX to use a given format either by supplying the **-layout** switch in the command line when launching ConfiX, or by supplying the `Layout` variable in the *\_Config* header, e.g.:

```
Layout = TABs
Layout = RadioLeft
```

The following layouts are currently available:

- |   |                     |
|---|---------------------|
| 0 | Icon tabs           |
| 1 | RadioLeft           |
| 2 | RadioRight          |
| 3 | Wizzard             |
| 4 | Icon & text buttons |
| 5 | Menu                |
| 6 | Text tabs           |

Please refer to chapter 1 (Introduction) for some example images.

#### Remarks:

1. Please note that you should leave the user in control of the selection for a particular layout whenever possible. That way, the user can set the overall appearance of the software on his computer, whilst maintaining a consistent user interface. In some situations however, you may want to force your program to use a specific layout, e.g. to ensure the software matches the user manual.
2. You may override the user's settings by supplying the `Layout` variable in the *\_Config* file (as described in this section).
3. You may override all of the previous settings by supplying the command line switch **-layout <name>**.
4. Whenever you've defined sub-groups in the *\_Config* file (see paragraph 5.2) the default layout (graphical TABs) will always be used, regardless any other settings.
5. The window height will automatically be adjusted to fit the largest configuration window.

## 5.2 Groups

Groups are identified in the same way as in the *Config* file, by enclosing the group name in square brackets. e.g.:

```
[General]
[Serial]
[Printing]
```

Additionally, a system of groups and sub-groups may be used to introduce a second row of TABs at the bottom of the window. To achieve this, you should declare a **Group** first, by entering a standard group name. Next you add the **sub-groups** that belong to this group, by specifying the **Group**-name followed by the **sub-group** name (separated by a comma). Like this:

```
[Ext]
[Ext,GPI]
[Ext,MPEG]
[Ext,DVD]
```

The first item in the example declares the group and hence will add a TAB in the top row. Whenever that TAB is selected, a new row of TABs will appear at the bottom of the window, showing three more TABs, called GPI, MPEG and DVD respectively.

Whenever using sub-groups, the default layout (Graphical TABs) will be forced into use, as it's the only layout capable of showing sub-groups correctly. If you don't want this behaviour, you may turn this feature off (from version 0.77 onwards).

## 5.3 Group header

The group name is generally followed by a set of variables, describing the group. The following variables may be used here:

### Icon

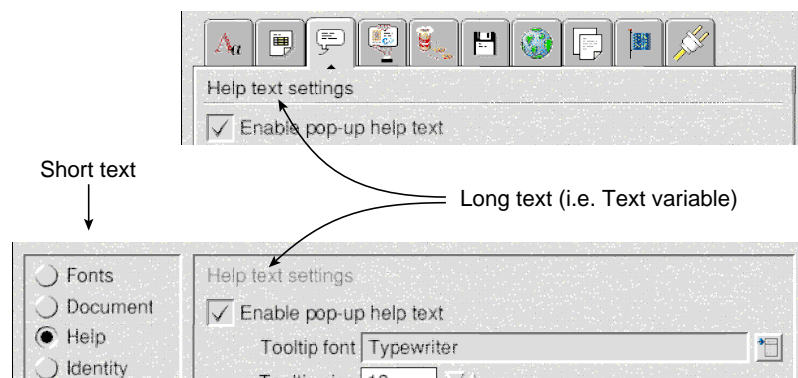
This is the name of the icon that should be displayed in the TAB for this group. The icon name should match one of the sprites supplied by your program in the *\_Tabs* file.

### Text

This is the text (or name) for each TAB. It will be displayed immediately below the selected TAB or inside the Help-line whenever the user moves the mouse pointer over the TAB.

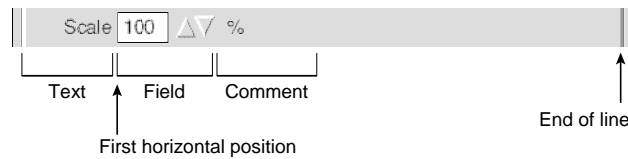
### Short

This is a short alternative for the **Text** above which will be used in preference when using radio buttons rather than the graphical TABs (i.e. one of the alternative layouts). If the **Short** variable is omitted, the contents of the **Text** variable will be used instead. Whenever the **Text** variable hold a longer string (e.g. more than one word), you are recommended to also supply the **Short** variable.



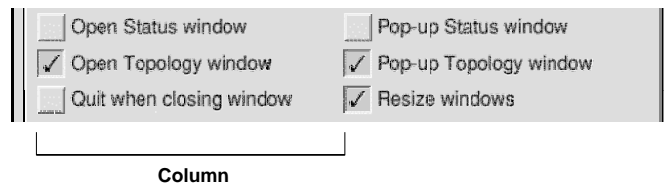
## Width

This variable gives the width of the leftmost part of an icon inside a configuration window. All of the icons will lineup to this position. If the variable is omitted, a sensible default is used instead. The position referred to by this variable is also called the *first horizontal position* (explained later).



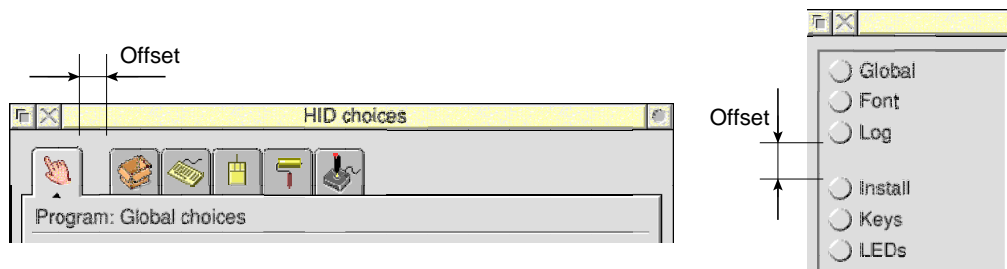
## Column

This variable is optional and specifies the width of the first column. By default the column width is used as specified in the !Config header. The Column variable in the group header may over-ride this value, for this particular group only. The next group will use the default value again (as specified in the \_Config header).



## Offset

By default all TABs are aligned neatly next to each other. However, you may want to insert a small gap, e.g. to create two sets of TABs. This can be done by specifying the (optional) **Offset** variable in the Group header. The default value of this variable is 0 (zero), which means no additional spacing. A value of, say, 40 OS-units, means that a gap of 40 OS-units will be visible between the previous TAB and this one.



For the default layout (i.e. the graphical TABs shown in the example above) this means that a horizontal space is inserted between two TABs. For some other layouts, e.g. the ones with the radio buttons, this means that a vertical space will be inserted between two radio buttons.

## Help

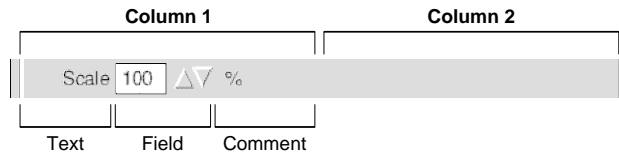
Whenever this variable is present, *Help*-button will be added to the window automatically (available from version 0.77 onwards). The parameter should be a path pointing to the required help file. This path will be called by issuing a **Filer\_Run** command when the *Help*-button is clicked. This can be very useful for Text and HTML files. Some examples:

```
Help=<MyApp$Dir>.Help.More.About
Help=<MyApp$Dir>.!Help
```



## 5.4 Object descriptors

Object descriptors are the most crucial part of the `_Config` file. All icons, variables, menus, tick boxes, etc. are defined this way. Each group contains a number of these objects which, together, will construct a single configuration window. By default each new icon will be placed on a new 'line' inside the window, unless specified otherwise. A typical icon will consist of a text, a field and optionally a comment; like this:



Each object descriptor should be enclosed in curly brackets '{' and '}' and each item should start on a new line. An example:

```
{
  Name = AutoOpen
  Text = Open window
  Type = Option
  Default = on
  Help = Open the main window when the program is started
}
```

The '{' marks the start of the object descriptor and the '}' closes the structure. Inside the structure, a set of variables is used to define the descriptor. The above example defines an option button (tick box) which is **on** by default. The text 'Open window' will be displayed next to the button and the variable **AutoOpen** will be generated in the `Config` file. The resulting window will look like this:



By default, each new icon will start on a new line in the window. This works much in the same way as a type-writer. After an icon has been placed inside the window, the 'cursor' will move to the start of the next line. If you don't want this to happen, you may place a markup character behind the '}' sign. E.g. placing a ';' (semicolon) after the '}' forces a move to the next column rather than to the next line. The next object, defined by the following descriptor, will then be placed on the same line, in the next column. The following markup characters may be used:

;	semicolon	Move to next column.
:	colon	Don't move (next icon will be placed immediately behind the current one).
-	minus-sign	Move to the start of the current line.
	vertical bar	Move to the first horizontal position on the current line (i.e. the left).

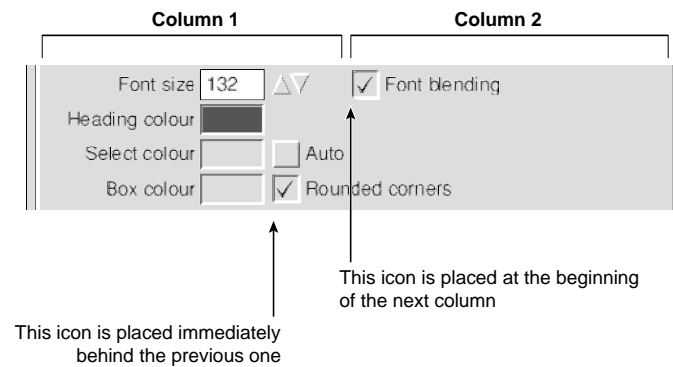
To following example will generate two tick boxes side by side on the same line.

```
{
  Name = AutoOpen
  Text = Open window
  Type = Option
  Default = on
  Help = Open the main window when the program is started
};
{
  Name = AutoClose
  Text = Close on Escape
  Type = Option
  Default = off
  Help = Close the main window when Escape is pressed
}
```

In this case the result will look like this:



The difference between using a ';' and a ':' is best illustrated by the following example:



The above is the result of the following code:

```
{
  Name = FontSize
  Text = Font size
  Type = Integer
  Width = 100
  Help = Font size in RISC OS font-units
  Default = 132
};
{
  Name = FontBlending
  Text = Font blending
  Type = Option
  Default = on
}
{
  Name = TColour
  Text = Heading colour
  Type = Colour
  Width = 100
  Help = Colour used for the heading background
  Default = 153,68,102
}
{
  Name = SelCol
  Text = Select colour
  Type = Colour
  Width = 100
  Help = Background colour for a selected item
  Default = 255,209,209
}:
{
  Name = SelAuto
  Text = Auto
  Type = Option
  Help = Calculate Select colour from heading colour (below)
  Default = off
}
{
  Name = BoxCol
  Text = Box colour
  Type = Colour
  Width = 100
  Help = Background colour for the 'Device' box
  Default = 232,232,232
}:
{
  Name = Rounded
  Text = Rounded corners
  Type = Option
  Default = on
  Help = Show rounded corners in Topology window
}
```

The following variables may be used to create an object descriptor. Please note that not all variables are needed or required in all descriptors. We will first define the most common variables and then describe each **data type** in more detail. All other variables are data type dependent and will be described in the relevant section.

### **Name**

This is the name of the variable, i.e. the variable name used inside the Config file. Please note that the same variable name can be used again in another Group. Your application should treat those as *different* variables.

### **Text**

This is the text to be displayed next to the icon. In the example it will show 'Open window' behind the tick box.

### **Comment or Legend**

This allows additional text to be added to an icon. The place where the additional text appears depends on the data type used. E.g. a comment will be placed *before* a tick box, but behind a text string.

### **Type**

This variable defines the data type. Data types can be e.g. Option, Radio, String, Menu, Font, etc. Data types are described in more detail in the next chapter.

### **Default**

This describes the default value for the variable. This value is used when !ConfiX creates a *Config* file for the first time and also when the user clicks the *Default* button.

### **Help**

This (optional) variable is used to supply an additional line of text that will be displayed inside the Help-line whenever the user moves the mouse pointer over the particular icon inside the window.

### **TextAlign**

By default the text in a readable or writable icon will be left aligned. If required, you may force an alternative alignment for the text though. E.g.:

```
TextAlign=Centre  
TextAlign=Right
```

### **Width**

By default the width of an icon will be extended to the rightmost edge of the window. If you don't want this to happen, you may specify a limited width in OS-units instead. Please note that, for a writable icon, this will only define the icon width, not the number of characters allowed inside it (the latter is defined by the *Size* variable). Some examples:

```
Width=300  
Width=250
```

### **Size**

This variable can be used to limit the number of characters that can be typed into a writable icon, such as a string or an integer. E.g. setting the size to 4, will only allow upto four characters to be entered in that field. E.g.

```
Size=24           (this will allow 24 characters to be entered in the writable field)
```

### **Menu**

This is a very powerful variable, used by the data types *Menu* and *Radio*. It allows flexible menus to be created as well as a menu of directory items (the latter may be useful if you want the user to select a file from a directory). A more detailed description will be given in the relevant section on Menus.

### **Enable**

This variable allow an icon to be greyed out depending on the state of another item. This mechanism is recursive and can be used accross TABs. More information can be found in paragraph 5.5. An example:

```
Enable = [GroupName]VarName,2
```



## 5.5 Greying out icons or groups of icons

When designing complex ConfiX windows, you may want to disable certain icons, depending on the state of another icon. A powerful feature is available from version 0.46 onwards, that allows you to link the state of an icon to the value of another one. This feature works across all TABs and allows nesting. In other words: The availability of a selection or item, may be controlled by the state of another icon, which in turn may be depending on another one.

The feature to grey out icons can be controlled by options (tick boxes), radio buttons and menus, allowing very powerful combinations to be made.

By default an icon is always available (i.e. enabled). However, if the `Enable` variable is present inside the definition of the *current* object, the value of the *specified* object will be inspected and compared to see if the *current* icon should be enabled. One of the following syntaxes can be used:

```
Enable = <varname>,<value>[,<value>[...]]
Enable = [<groupname>]<varname>,<value>[,<value>[...]]
```

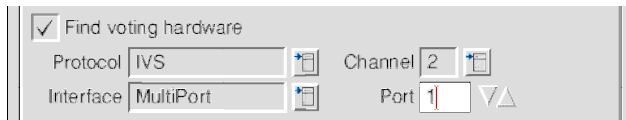
No spaces are allowed in the part behind the '=' sign. Some examples:

```
Enable = Open,1
Enable = [Video]WideScreen,1
Enable = [Main]ShowLevel,2
```

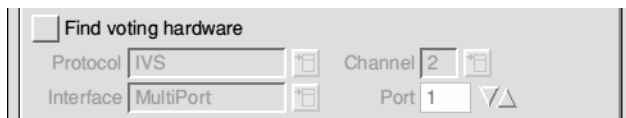
If the value of the specified object is equal to the value specified after the comma, the icon will be enabled and can be accessed by the user. If the value doesn't match, the icon will be greyed out.

When using two rows of TABs (one at the top and one at the bottom of the window) group names will be in the form of:

[Main,Hardware]



The option *Find voting hardware* is turned on and the icons used to set the hardware configuration are available.



Here the option is turned *off* and all icons linked to it, are greyed out.

The above would be generated as follows:

```
{
  Name = Enable
  Text = Find voting hardware
  Help = Allow use of voting hardware on this machine
  Type = option
  Default = off
}
{Name = Protocol
  Text = Protocol
  Type = Menu
  Help = Voting system protocol
  Default = IVS
  Menu = #Protocol;dir;DeviceDrv:Modules.000B
  Width = 300
  Enable = Enable,1
};
```

etc.

Instead of specifying the condition(s) to **Enable** an object, it is also possible to specify the condition(s) to **Disable** an object.

```
Disable = <varname>,<value>[,<value>[...]]
Disable = [<groupname>]<varname>,<value>[,<value>[...]]
```

No spaces are allowed in the part behind the '=' sign. Some examples:

```
Disable = Volume,0
Disable = [Video]WideScreen,0
Disable = [Main]ShowLevel,5
```

If the value of the specified object is equal to the value specified after the comma, the object will be disabled and can no longer be accessed by the user. If the value doesn't match, the object will be available for use, subject to the grey-out state of the parent object.

When using two rows of TABs (i.e. netsted groups), group names will be in the form of:

```
[Main,Hardware]
```

The example below illustrates the use of both **Enable** and **Disable**. In this case, the grey-out state of the lower three objects are controlled by the first one (**Enable**). The lower two are dependent on the second one. The difference between these two is that **StoragePeriodUnits** is disabled when **StoragePeriod** has reached a value of 0, whilst the bottom one (type **Comment**) is enabled.

```
{
  Name = Enable
  Text = Enable logging
  Type = Option
  Default = off
  Help = When on, some actions will be logged
}
{
  Name = StoragePeriod
  Text = Keep
  Type = Integer
  Default = 3
  Size = 3
  Width = 112
  Help = Log files are kept for this period
  Enable = Enable,1
}:
{
  Name = StoragePeriodUnits
  Type = Menu
  Default = 1
  Menu = Period;Same as for
create|,0;Day(s),1;Week(s),2;Month(s),3;Year(s),4
  Disable = StoragePeriod,0
}
{
  Type = Comment
  Legend =
  Text = Keep indefinite
  Enable = StoragePeriod,0
  Options = -col 11
}
```

---

## 6 Data types

---

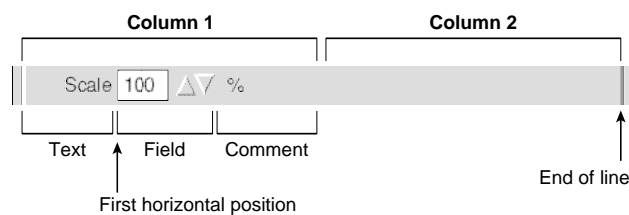
This chapter describes each data type in more detail. In particular, it defines all variables necessary to create a variable descriptor for this data type. For certain data types alternative spelling is allowed to make it easier for a programmer to remember. Data types are defined by the `Type` variable inside a variable descriptor. E.g.

```
Type=Option  
Type=String  
Type=URL
```

At present the following data types are recognised:

- 1 Option, Boolean, Switch, TickBox
- 2 String, Text
- 3 Integer, Numeric, Number, Value
- 4 Float, FloatingPoint, Real
- 5 Menu
- 6 Radio, RadioButtons
- 7 Font, Typeface
- 8 Colour, Color
- 9 Ruler, RuleOff, Divider
- 10 URL, FTP, WebSite, Web, Link, HTTP, Internet
- 11 Button
- 12 Icon, Sprite
- 13 OLE
- 14 IP Address
- 15 BlockDriver
- 16 Comment
- 17 Info
- 18 Slider, Fader
- 19 Spacer
- 20 Caption

In the following paragraphs each of these data types will be further explained. The diagram below can be used as a guideline to the terminology used in the descriptions.



Data types are defined by a variable descriptor as explained in the previous chapter. A variable descriptor consists of a number of variables that control the appearance of the icon. Please note that most variables are optional; if you don't supply then, sensible defaults will be used instead. When describing an icon however, some variables are mandatory and some are recommended.

The minimum set of variables to create a useful icon is:

<b>Name</b>	This is the name used for the variable in the <i>Config</i> file.
<b>Text</b>	This is the text to be placed in the icon.
<b>Type</b>	This is the data type

Additionally, we strongly recommend the use of the following variables (if possible):

<b>Default</b>	Supply a sensible default, so that the user can revert to the default settings.
<b>Help</b>	Supply a useful description of the field, so that the user won't need a manual.

Empty lines may be inserted by inserting an 'empty' descriptor:

```
{  
}
```

The `Height` variable may be used to define the space taken by the 'empty' descriptor, e.g.

```
{  
  Height = 12  
}
```

## 6.1 Option, Boolean, Switch, TickBox

This is probably the simplest form of a configuration option and is commonly used to turn options **on** or **off**. A typical descriptor looks like this:

```
{
  Name = AutoOpen
  Text = Open window
  Type = Option
  Default = on
  Help = Open the main window when the program is started
}
```

And the result will look like this:



If you want to lineup the icon with other icons, e.g. a string, you may force it to be placed at the *first horizontal position*, by adding an empty `Comment` variable to the descriptor, e.g.:

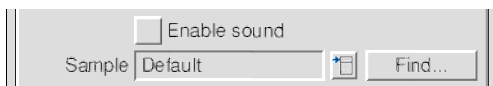
```
{
  Name = Enable
  Text = Enable sound
  Comment =
  Type = Option
  Default = off
  Help = Turn sound effects on/off
}
```

Of course the `Comment` variable may be used to insert some additional text before the icon, e.g.:

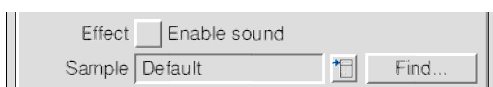
```
{
  Name = Enable
  Text = Enable sound
  Type = Option
  Comment = Effect
  Default = off
  Help = Turn sound effects on/off
}
```



Without Comment variable



With 'empty' Comment variable



With Comment variable

## 6.2 String, Text

In some situations the user may need to enter a text such as a filename, the name of a person, etc. The data type **String** allows any ASCII character to be entered, including numbers, spaces and special characters. By default a maximum string length of 255 characters is assumed and the icon will expand to the right of the window, e.g.:



The above icon is produced from this descriptor:

```
{
  Name = User
  Text = Name
  Type = String
  Default =
  Help = Enter your name here
}
```

You may however want to limit the number of characters allowed in the string to, say, 20 characters and at the same time limit the space used by the string in the window. This can be done by adding the variables **Size** and **Width**. *Size* will define the maximum number of characters allowed in the string, whilst *Width* controls the width of the icon in OS-units, e.g.:

```
{
  Name = User
  Text = Name
  Type = String
  Default =
  Help = Enter your name here
  Size = 20
  Width = 300
}
```

will produce this:



### Dropping files and directories into a string

The extra variable **Drop** may be added to allow files or directories to be dropped straight into the writable field. The **Drop** variable may take the following values:

<b>Drop=path</b>	If a file or directory is dropped in the writable field, the entire path is used.
<b>Drop=leaf</b>	Same as the above, but only the leafname will be stored inside the icon.

### Switches

Additionally, a number of switches may be added to the value. Switches are always preceded by a '-' sign.

#### **-strip <path>**

This will cause the <path> to be stripped from the beginning of the full pathname before putting the result in the icon.

#### **-url**

This will cause the resulting filename (in the icon) to be converted from/to URL conventions, i.e. a '.' (full stop) will be converted to '/' (slash) and vice versa.

*From version 0.84 onwards*, this option also adds the prefix 'file:/' to the beginning of the name, when converting to URL format. Furthermore, a '/' is added to the end of a URL if it is a directory rather than a file.

#### **-sub <directory>**

Add a further subdirectory to the path that is dragged in. It also verifies whether the path exists. If it doesn't exist, the dragged path will be ignored. *This feature was added in version 0.84.*

#### **-dir**

Allow only directories to be dragged in. *This feature was added in version 0.84.*

### **-file**

Allow only files to be dragged in. *This feature was added in version 0.84.*

### **-ext**

Add a DOS extension to the filename automatically. This can be particular useful when creating web resources that are stored on the RISC OS filesystem without the equivalent DOS extension.

### **-setext <ext>**

This option will force the DOS extension to the one specified in <ext>.

### **-fix**

The size of the icon is fixed. Adding the optional Menu-icon will not affect the icon size as usual.

### **-fs**

Show only the *Filing System* part of the path dropped in the icon.

E.g. if a file was called **ADFS::4.\$MyFile**, the result would be **ADFS**.

### **-add <separator>**

This option allows multiple selections to be made in the same icon. E.g. when dropping a number of files in the icon, their names (full, leaf or fs) will be added to the one(s) already there, separated by a configurable character.

### **-remove**

If this option is added, selecting an item that is already present in the string, will remove that item from the string. This allows a menu to be added to the string, which can be used to both add and remove an item.

### **-pwd <character>**

If the *String* field is used to enter, say, a password, you may want to hide any text entered into the field. If such is the case, supply the **-pwd** switch, followed by a character that you want to use as a replacement. E.g.: if you supply the switch **-pwd \*** and the user enters 123456, this will appear as **\*\*\*\*\***.

### **-nospace**

This switch can be used to avoid spaces in text objects. If the switch is present, a space will automatically be converted into a hard-space (RISC OS 5). Please note that the password option (-pwd) implies a **-nospace** automatically. The **-nospace** switch was added in version 1.10.

Some examples:

```
Drop = path -strip <MyApp$Input>.Pages -url -ext
Drop = leaf -fs -add , -remove
```

Please note the comma after the **-add** switch, which specifies the separator.

## **Adding a menu to a string icon**

To make the string icon even more powerful, a menu may be added to allow the user to select a sensible default from a list of preferred settings, rather than typing directly. To do this, add a **Menu** variable to the descriptor, e.g.:

```
{
  Name = Priority
  Text = Iconbar
  Type = String
  Width = 264
  Size = 9
  Default = &0E000000
  Menu = Priority;Anywhere|,0;Left of Display manager,&0E000000;Right of Display manager,&2E000000
  Help = Position priority of icon in the iconbar
}
```

This will produce the following in the configuration window:



Please note that in this example, a *string* icon is used to enter a *hexadecimal* numerical value. The string will be stored in the *Config* file 'as is', so it is up to your application to translate this value into something useful for your program.

The **Menu** variable uses the following syntax:

```
Menu = <title>;<item_1>[ | ],<value_1>;<item_2>[ | ],<value_2>...
```

Alternatively, filer menus may be added to allow e.g. items from a directory to be selected. For a more detailed description of the Menu variable, please refer to **chapter 6.5: Menus**.

```
Menu = #<title>;<type>;<parameter>
```

e.g.:

```
Menu = #Sample;file;<MyApp$Dir>.Sound  
Menu = #Skin,dir;<MyApp$Dir>.Skins  
Menu = #FilingSystem;fs;-ignore default
```



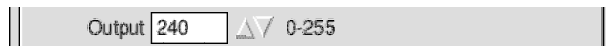
## 6.3 Integer, Numeric

Rather than entering a string, you may wish to restrict the input to numerical values only. Data type *Integer* can be used to create a numerical field in a configuration window. This will allow only numbers to be entered in the field. Furthermore the *Width* and *Size* of the field may be controlled by the appropriate variables.

In the following example, we'll try to create an icon to set the maximum volume of the sound system. The value entered should be in the range 0-255. We therefore specify a field size of 3 (a maximum of 3 character can be entered). Additionally, the Integer field allows us to specify a minimum and a maximum value. !ConfiX will check for these limits, so you won't have to do this in your program. The following example will create such a field:

```
{
  Name = MaxOutput
  Text = Output
  Type = integer
  Default = 255
  Comment = 0-255
  Min = 0
  Max = 255
  Step = 1
  ShiftStep = 10
  Width = 120
  Size = 3
  Help = Output sound level
}
```

In the configuration window it will look like this:



To the right of the writable icon, two arrows will be inserted to allow control of the numerical value with the mouse. Clicking the **up** arrow will increase the value, whilst the **down** arrow can be used to decrease the value. An extra variable, *Step*, may be added to allow flexible control over the steps the value takes with each click on one of the arrows. If no *Step* value is given, the value will increment or decrement by 1, else the specified value is used. From version 0.54 onwards a further variable *ShiftStep* has been added which allows a different step to be applied when the *Shift* key is depressed whilst clicking the arrows. If the variable is omitted, it will default to  $10 * \text{Step}$  (10 times the *Step* value).

### Comment

To help the user to decide what a sensible value would be, it might be considered to add both limits of the value in the *Comment*-part of the field, as shown in the example. Furthermore, the *Help*-line may be used to explain the usage of the field. It is also possible to specify an alternative comment, e.g.:

Comment=minutelminutes

The comment is added to the object as a suffix (like the *units* in a measurement). If the number in the field is '1' it will display the first comment. In all other cases it will display the 2nd one. If only one comment is given, it will always be displayed. When present, the two comments should be separated by a vertical bar '|'.

The following options may be added by specifying the appropriate switch to the *Options* variable:

### -noarrows

If you don't want the *up* and *down* arrows to appear to the right of the numerical field, you should specify the switch **-noarrows** in the *Options* variable. This can be particular useful when entering, say, a registration number, for which it is not appropriate to adjust it with up/down control.

### Example

```
Options = -noarrows
```

## 6.4 Float, FloatingPoint, Real

---

Floating point fields allow any numerical value to be entered, including a decimal point. Apart from this difference, floating point fields behave the same as *Integer* fields (explained in the previous paragraph).

As this field is used to enter floating point variables, you may want to define a precision (i.e. the number of decimals after the decimal point). By default, the numerical field will inherit the precision from the `Step` value. E.g. if you've defined a `Step` of 0.01, the value entered in the writable field will have a precision of 2 decimal places (e.g. 38.24). If the `Step` size doesn't contain a decimal point (e.g. 2), a precision of 3 decimal places is assumed.

Any other precision can be forced by specifying it in the `Options` variable, using the **-dec** switch, e.g.:

```
Options = -dec 2
```

### Options

The following options may be added by specifying the appropriate switch to the `Options` variable:

#### **-dec <precision>**

By default, the precision of the floating point number (i.e. the number of decimal places after the decimal point) will be equal to the number of places specified in the `Step` variable. If you don't want this behaviour, you may specify the number of decimal places explicitly, by adding the **-dec** switch, followed by the required precision, to the `Options` variable.

#### **-noarrows**

If you don't want the *up* and *down* arrows to appear to the right of the numerical field, you should specify the switch **-noarrows** in the `Options` variable.

### Example

```
Options = -dec 3 -noarrows
```

## 6.5 Menus

This is one of the most complex yet powerful data types used by !ConfiX. It can be used to create flexible user menus, but also to generate a menu showing the contents of a directory. First we will look at the standard menus.

### 6.5.1 Simple menus

When creating a menu, we need to be able to define a title for the menu, a text for each entry and a value for each entry that will be used whenever that entry is selected. Values can be both numerical and alpha-numerical to allow flexible menus to be created. A menu is defined by the **Menu** variable on a single line. Each item is separated by a ';' (semi-colon) and within an item, the text and value are separated by a ',' (comma). A '|' sign (vertical bar) can be inserted at the end of an item to generate a dotted line *below* this item. Items can be greyed out by inserting a '~' (tilde) at the beginning of the item. The syntax is as follows:

```
Menu = <title>[~]<item_1>[ | ],<value_1>;<item_2>[ | ],<value_2>...
```

#### Special characters

	Vertical bar	Insert a dotted line <i>below</i> this time.
~	Tilde	Grey-out this item.
,	Comma	Separator between item text and value.
;	Semi-colon	Separator between menu items.

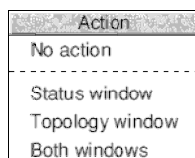
Let's consider the following example:

```
{
  Name = Select
  Text = Mouse Select
  Type = Menu
  Help = Action when iconbar clicked with Select
  Menu = Action;No action|,0;Status window,1;Topology window,2;Both windows,3
  Default = 2
  Width = 400
}
```

This will produce an icon that looks like this:



Clicking the menu-icon will reveal the following menu:



Please note the vertical bar (|) in the Menu definition after the first item (No action). If this vertical bar is present at the end of the entry, it will result in a separator in the menu (the dotted line).

The first item in the Menu definition is the menu-title 'Action'. The next items show the various menu entries with their values (separated by a comma). Clicking the first menu item (No action) will result in a value of 0 to be used and so on. The result in the *Config* file will be something like this:

```
Select=2
```

Please note that the <value> in the menu definition can be alpha-numerical. Rather than specifying a number (e.g. '2') you may specify a textual value, e.g. 'left'. Here is an example:

```
{
  Name=Align
  Type=Menu
  Menu=Align;Left hand side,left;Centre of the screen,centre;Right hand side,right
  ...
}
```

If the second item was selected, it would result in the following line (in *Config*):

```
Align=centre
```

## 6.5.2 Directory menus

In some cases it may be useful to create a menu of directory items, e.g. if the user needs to select a sound sample from disc. Such menus can be easily created by placing a '#' sign (hash) at the beginning of the Menu definition. The '#' sign is followed by the items *Menu-title*, *type* and *path*, all separated by a ';' (semi-colon). The syntax is as follows:

```
Menu = #<title>;<type>;<parameter>
```

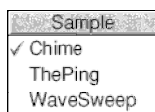
The second descriptor in the next example will create a directory menu:

```
{
  Name = SoundNew
  Text = New device detected
  Type = Option
  Default = on
  Help = Play sound sample when new USB device detected
};
{
  Name = SampleNew
  Text = Sample
  Type = Menu
  Menu = #Sample;file;<MyApp$Dir>.Sound
  Default = Chime
  Help = Select a sound sample to play
  Width = 350
}
```

In the configuration window, it will look like this:



Clicking the menu-icon will reveal a menu that shows all in the given directory, e.g.:



Please note that the second item on the Menu line in the the above example is **file**, which indicates that the menu will only show any *files* found in that directory. Any *subdirectories* will be ignored. Alternatively, you may only display any *subdirectories* by specifying the type **dir**.

The following types and their alternative spellings are recognised:

type	name	alias	Further info
1	file	file, files	
2	dir	dir, directory, map, folder	
3	fs	filingsystem, filesystem, fileswitch	See paragraph 6.5.3 for a detailed description.
5	tt	tt, truetype	See paragraph 6.5.4 for a detailed description
6	ext	external, indirected, fromfile	See paragraph 6.5.5 for a detailed description
7	print	pr, printer, printers	See paragraph 6.5.6 for a detailed description

### Context sensitive menus

A menu can be made context-sensitive by using system variables. For this, you need to specify a prefix for a range of system variables, by defining the 'SysVar' variable in the header of the \_Config file. See paragraph 5.1.2. for more information on the use of SysVar. For example, you could use something like:

```
Menu = #Demo;file;<MyApp$Dir>.Files.Menu.<MyVars$Values$Group@Value>
```

in which **<MyVars\$Values\$Group@Value>** is the system variable set by another variable. It can be used to open another menu, depending on the state or value of the specified variable. System variables can only be used when the SysVar variable in the header is defined. *This features is available from version 0.85 onwards.*

## Switches

The following *switches* may be added to the end of the 'Menu' line:

### **-ignore <filename>[,<filename>]**

Don't show the specified name in the menu. Multiple filenames should be separated by commas and there should be no spaces between the items. For example:

```
Menu = #Language;file;<MyApp$Dir>.Languages -ignore UK,NL
```

## Options

The variable **Options** may be used to further process the selected menu entry. The following switches are recognised and multiple switches are allowed:

### **-prefix <text>**

A prefix may be added to the selected menu entry, e.g. a file prefix.

### **-suffix <text>**

In addition to the prefix, it is also possible to add a suffix to the selection, e.g. a filename or extension.

### **-none <Default>**

If this switch is present, an extra option will be added to the top of the menu, that can be used to clear the value of this variable. The text following the **-none** switch will appear in the menu ('<Default>' in the above example). When selected, the value of the variable using this switch, will be blank, but it may cause an extra tick in the menu, if it matches any of the other menu items.

### **-default <Default>**

Same as above (the **-none** option) but this time <Default> will be used as the default value, rather than clearing the variable.

### **-noext**

If this switch is present, any DOS extension will be stripped from the filename.

### **-add <separator>**

This option allows multiple selections to be made in the same icon. E.g. when dropping a number of files in the icon, their names (full, leaf or fs) will be added to the one(s) already there, separated by a configurable character.

### **-remove**

If this option is added, selecting an item that is already present in the string, will remove that item from the string. This allows a menu to be added to the string, which can be used to both add and remove an item.

An example to illustrate the use of **Options**:

```
Options = -none Default -noext -add , -remove
```

## Remark

1. No spaces are allowed in the suffix or in the <default> fields above, as spaces are used to separate the options. If you need spaces, use hard-space instead, by typing Alt-SPACE.

### 6.5.3 Creating a filing system menu

When using filing system menus (fs) the name -and optionally the numbers- of all available filing systems will be shown in the menu. In this case the `Menu` variable would be something like this:

```
Menu = #Title;fs;
```

and the resulting menu will look like the one on the left. The corresponding filing-system-numbers may be added as a prefix, by specifying the `-numbers` switch:

```
Menu = 'Title;fs; -numbers
```

The list of items can be restricted to a given range, specified by the `-allow` switch. Furthermore, items can be removed from the list, by specifying them in the `-ignore` switch.

FilingSystem	FilingSystem	FilingSystem
None	None	None
-----	-----	-----
ADFS	8 ADFS	ADFS
null	13 null	CDFS
printer	14 printer	Share
vdv	17 vdv	
rawvdu	18 rawvdu	
kbd	19 kbd	
rawkbd	20 rawkbd	
CDFS	37 CDFS	
Resources	46 Resources	
Pipe	47 Pipe	
devices	53 devices	
Share	99 Share	
JetDirectFS	144 JetDirectFS	

#### Switches

Some fs-specific *switches* may be added to the end of the 'Menu' definition, after the last semi-colon, to allow a more flexible control. The following switches are available:

##### **-ignore <fsnumber>[,<fsnumber>,<fsnumber>...]**

This switch allows certain filing systems to be omitted from the menu, so that they cannot be selected by the user. By default all available filing systems will be shown, including some less useful ones, such as *vdv*, *keyboard*, *null*, etc. Filing Systems must always be specified by their FS number. For a full description on all Filing System numbers, please refer to the PRM, page 2-19. The filing system numbers are separated by commas and no spaces are allowed. An example:

```
Menu = #FilingSystem;fs;-ignore 2,3,18,23,44
```

This will exclude the specified filing systems from the menu. Rather than specifying a series of filing systems, you may use the default exclusion set, by specifying `default` after the `-ignore` switch, like this:

```
Menu = #FilingSystem;fs;-ignore default
```

The advantage of the method is that any new filing system (that might be added to the OS at any time) will be available immediately without altering your code. See the rightmost example image above.

##### **-allow <fsnumber>[,<fsnumber>,<fsnumber>...]**

Rather than specifying the filing systems that should NOT be shown in the menu, you may explicitly define the filing systems that you DO want to appear in the menu. All other filing systems will be ignored. Again, you need to enter the filing system numbers, separated by commas. E.g.:

```
Menu = #FilingSystem;fs;-allow 8,18,23,99
```

Alternatively, you may want to use the default set of allowed filing systems, by specifying `default`, e.g.

```
Menu = #FilingSystem;fs;-allow default
```

Please note that the `-allow` switch may ONLY be supplied if the `-ignore` switch is NOT used. If both are supplied, only the `-ignore` switch will be used.

##### **-numbers**

This switch should be added if you want filing system numbers to be selected, rather than the filing system names as in the second example image above, e.g.

```
Menu = #FilingSystem;fs;-ignore default -numbers
```

#### 6.5.4 TrueType Font menu

Most applications running under RISC OS will use the Outline Font Manager to display text. Selecting one of the available fonts from ConfiX is usually done by using the data type Font (see paragraph 6.7). Some applications, such as Oregano 2 however, are using TrueType fonts for this purpose. As in most cases there is no relationship between the file names and the actual font names, we can't use the standard directory menu here.

For this purpose the menu object has been extended to support TrueType font files. ConfiX will scan each file and directory from the given path for valid fonts, and extract the font family name from within the file. Fonts can be grouped in two ways and the font family name will be assigned accordingly:

1. If multiple files are available within a single font family (e.g. Bold, Italic, etc.), the font family name will only occur once in the menu.
2. If the family members are grouped into a family folder, only the first file will be checked for a family name.

Please note that extracting the names from the TrueType font files is a rather complex and time consuming operation, as the header of each file has to be read and parsed by ConfiX. The more files you have in your fonts-directory, the longer it takes to scan them. For this reason it is recommended to use the second method (i.e. place any family members into a family folder) if you can. Some programs (e.g. Oregano 2) are supplied with a conversion program, that automatically re-organises your fonts into family-folders. This can be particularly useful when porting fonts from another platform to RISC OS. Such a program is **!TT202**, which can be downloaded from:

`http://www.xat.nl/en/riscos/sw/oregano/`

The menu definition to create a TrueType font menu is rather simple:

```
Menu = #<title>;tt;<path>
```

For example:

```
Menu = #Font;tt;<MyApp$Dir>.Fonts
```

Please note that a delay may occur when first opening such a menu, as ConfiX will have to cache the font names. Once the names are cached, any further TrueType font menu, from the same path, will be opened instantly.

### 6.5.5 External menu

Creating a user-defined menu, as described in paragraph 6.5.1, can be rather limiting, as the entire menu definition has to fit on a single line which can't be longer than 256 characters. For this reason, external menus have been added, allowing the entire menu definition to be loaded from a text file on disc. The syntax for the menu variable is:

```
Menu = #<title>;ext;<path>
```

A typical example of an external menu is given here:

```
Menu = #Language;ext;<MyProg$Dir>.Files.Language
```

The rightmost part of the line is a path, pointing to the text file that contains the external menu definition. The text file consists of a number of lines, each representing a single menu entry. The syntax for a single line is:

```
<value>,<text>
```

Some examples:

```
3,Check modem
en-gb,English (UK)
```

The first part (before the comma) is the value that will be used to store in the Config file when selected. The rightmost part (after the comma) is the text that will be shown in the menu. Please note that this is exactly the other way around as for the standard user-defined menus.

#### Special characters

	Vertical bar	Insert a dashed-line <i>below</i> this item.
~	Tilde	Grey-out this item.
,	Comma	Separator between <text> and <value>

#### Dashed lines

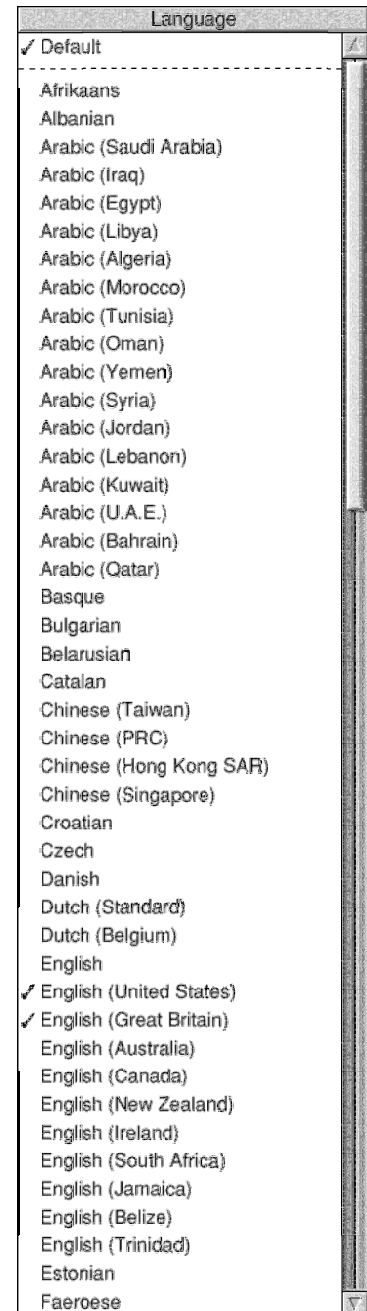
A 'l' can be added to the end of a line to make a dashed line appear in the menu below *this* item. E.g.:

```
en-gb,English (UK) |
```

#### Greyed out items

An item can be greyed out by inserting a '~' (tilde) at the start of the line, e.g.:

```
~en-gb,English (UK)
```



#### Options

In addition to the options described in paragraph 6.5.2, the following switch may be added to the Options variable:

##### -full

When this switch is present, the full line from the external text file will be used in the menu, rather than just the part after the comma.



### 6.5.6 Printer menu

Using printers under RISC OS, is generally done through !Printers. This program is part of the standard RISC OS software suite and offers the user a universal and intuitive manner to address multiple printers. Multiple printers may be connected to a single computer, and others can be accessed e.g. over a network. Whenever the user wants to print to a certain printer, clicking the printer-icon in the iconbar is sufficient to select the required printer. All subsequent printing will now be done on the selected printer.

A downside of this user-controlled mechanism is, that a software package generally cannot select a specific printer for a certain job. Imagine a situation where someone uses a business package that can print invoices and leaflets. The invoices must be printed on a specific printer (loaded with company stationary), whilst all other output should be printed on the standard office printer.

At present, RISC OS has no mechanism to allow an application to select a specific printer. In the examples (available from the ConfiX web pages) we've added a few BASIC procedures to allow a given (named) printer to be selected from your software. !ConfiX can be used to help the user to select a printer for each job.

Imagine the following situation, where three printers are configured in !Printers:



Each printer MUST have a different name. In this case they are called 'Brother', 'QMS2060' and 'ToFile'.

The syntax is as follows:

```
Menu = #<title>;printer;
```

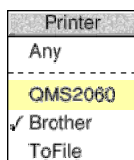
We could create a printer menu in ConfiX like this:

```
{
  Name = Invoices
  Text = Printer
  Type = Menu
  Menu = #Printer;printer;
  Options = -none Any
  Width = 400
  Help = This printer is used for printing invoices
}
```

which would produce the following line in the configuration window:



Clicking the menu now reveals the following menu:



The currently active printer (QMS2060 in this case) is highlighted, whereas the selected printer for this job is ticked. An extra option is added to the top of the menu by specifying the **-none** switch in the Options variable. The latter allows the variable to be cleared, so that your application won't attempt to select a printer.

### Options

The following switch may be added to the Options variable:

#### **-none <Default>**

This switch can be used to add an extra option to the top of the menu, allowing the user to de-activate the automatic printer selection mechanism. The text <Default> will be shown in the menu. In the above example the text 'Any' was used, to indicate that any printer can be used for this print job. Please note that the text <default> cannot contain spaces, as these are used as separators between options. If you must use spaces, use a hard-space instead (i.e. Alt-Space).

## 6.6 Radio, RadioButtons, Selector

Radio buttons provide a way to make an exclusive selection, i.e. only one button can be pressed at any time. They work much in the same way as Menus (described in the previous paragraph). The only difference is the way they are presented in the window. The reason for this, is that it makes it easier for the programmer to change a menu into a set of radio buttons or vice versa. The 'Menu' variable is used to define the buttons and their value.

The syntax is the same as for the Menu-object:

```
Menu = <title>[~]<item_1>[ ],<value_1>;<item_2>[ ],<value_2>...
```

First an example:

```
{
  Name = Mode
  Text = Display mode
  Type = Radio
  Menu = Mode;User mode,0;Advanced,1;Full info,2
  Stack = hor
  Width = 250
  Default = 0
  Help = Select default display mode
}
```

This will produce something like this:



The same definition, but with **Type=Menu**, would produce this:

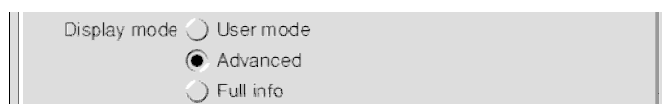


Please note that the menu-title isn't used here, but you should supply it anyway, to allow easy conversion from radio-buttons to menus.

### Stacking of radio buttons

Radio buttons can be stacked in two ways: horizontal and vertical. The buttons in the first example on this page are stacked in a horizontal array, which is defined by the line: **Stack=hor**.

Changing the definition to **Stack=vert** will produce the following set of icons:



The Stack variable will allow the following values:

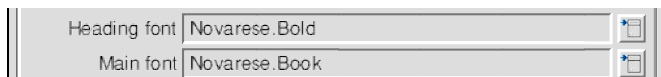
#	name	alias
0	vert	Vertical, TopDown, TopToBottom
1	hor	Horizontal, LeftRight, LeftToRight

## 6.7 Font, Typeface

It is often necessary to select a font from a configuration window. Although the font system under RISC OS works very well and is nicely integrated with the operating system, buffering font names, creating a menu and decoding a selection is quite a job. This data type takes the pain out of creating font menus. An example:

```
{
  Name = Heading
  Text = Heading font
  Type = Font
  Default = Homerton.Bold
  Help = Font used for headings
}
{
  Name= Font
  Text = Main font
  Type = Font
  Default = Homerton.Medium
  Help = Main font used in status window
}
```

This will produce two font menus, like this:



Please note that the icon is extended to the rightmost edge of the window, as the `Width` parameter is omitted. You may limit the width of the icon by supplying a suitable value for `Width`. Clicking the menu-icon to the right of the field, will open the font menu:



The font-menu is the same one as used in other RISC OS applications. Some fonts will have a further submenu to select any of its family members. The currently selected font (i.e. the one in the icon) will have tick displayed to the left of its name.

Selecting a font this way, will result in the following kind of variables in the *Config* file:

```
Heading=Homerton.Bold
Font=Homerton.Medium
```

## 6.8 Colour, Color

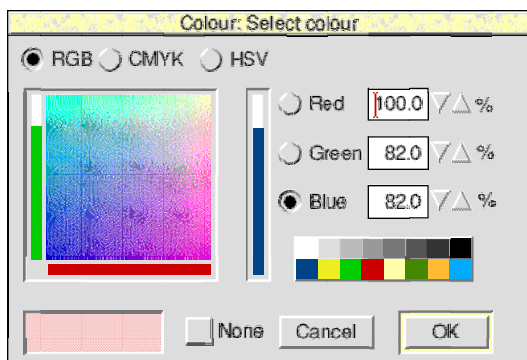
Another useful data type is the use of the standard RISC OS colour selector. The currently selected colour will be displayed immediately in the icon created this way and clicking the icon will open the standard RISC OS colour selector. The following descriptor:

```
{
  Name = BoxCol
  Text = Box colour
  Type = Colour
  Width = 100
  Help = Background colour for the 'Device' box
  Default = 232,232,232
}
```

will generate this type of icon:



The actual icon (the box to the right of the text 'Box colour') will show the currently selected colour. Clicking the box will cause the standard RISC OS colour selector to be opened:



The user may now adjust the colour using any of the standard colour models **RGB**, **CMYK** or **HSV**. Once satisfied with the new colour, click OK to confirm. The colour box in the icon will be redrawn to reflect the new colour.

Please note that it is also possible to select NO colour. To do so, select the tick box 'None' and click OK. The colour box in your icon will then look like this:



Instead of the selected colour, the box will now show a cross. A nice feature of !ConfiX is that the original colour is still remembered. So, if you untick the 'None' selector, the original colour will be restored.

### Default colour format

By default, the colour will be stored in the *Config* file in **RGB** format, using the following syntax:

```
<colour> = <red>,<green>,<blue>[,<transparency>]
```

The values for **red**, **green** and **blue** are always present, each separated by a comma. Each of the colour components should be in the range 0-255. Optionally, a **transparency** level may be stored in the range 0-255. As the current RISC OS colour selector only supports two levels of transparency (i.e. colour or no-colour), the transparency value will be either 0 or 255. Some examples:

```
BoxCol = 232,240,240
Background = 240,422,240,255
TextColour = 255,255,255
```

### Options

The following switches are allowed in the *Options* variable.

#### **-wimp**

This allows any of the 16 WIMP colours to be selected, rather than any colour out of the full 16M palette.

## Alternative colour formats

From !ConfiX version 0.86 onwards, a range of alternative colour formats is supported. These are serviced by the XML support module (eXML). For developers, a set of XML support libraries are available on request. !ConfiX can now read and interpret any valid colour format (see below) without the need to specify it.

When writing the colour back to the 'Config' file, you may specify the required format, either locally at the object level, or globally in the '\_Config' header. If nothing is specified, the (default) simple format (14) is used as before.

An alternative colour format is selected by including the Format variable in the object definition. For example, this object definition:

```
{
  Name = BackCol
  Text = Background
  Type = Colour
  Format = RGB
  Default = 255,100,92
  Width = 100
}
```

will result in the following colour notation:

```
BackCol = rgb(255,100,92)
```

Colour formats can be specified by their numerical value, or by their name. The following colour formats are currently supported:

#	Name	Description (example)
0	Default	Use the default format (as defined in the header)
1	RISCOS	&FF13D400
2	RGB	rgb(255,40,128)
3	RGB%	rgb(100%,22%,50%)
4	FullHex	#D413FF
5	ShortHex	#D1F
6	CMYK	cmyk(255,100,80,4)
7	CMYK%	cmyk(100%,80%,64%,2%)
8	HSL	hsl(360,100%,100%)
9	HSV	hsv(360,255,255)
10	HSV%	hsv(360,100%,100%)
11	Name	Named colours, such as: white, yellow, green, etc.
12	YUV	not yet implemented
13	YUV%	not yet implemented
14	Simple	Simple RGB format without any prefix, e.g. 255,40,64 or 255,40,64,32

For a full description of the various colour formats, please refer to the eXML API, available from us on request. The colour format can be defined globally (i.e. for all objects) by specifying it in the ColourFormat variable in the '\_Config' header.

When writing back the colour in the 'Config' file, the required colour format will be determined in the following order of priority:

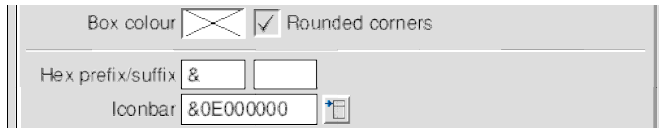
1. When the header variable ColourAuto=1, the colour format is inherited from the original value in the 'Config' file. Each object can thus have its own format without the need to specify it.
2. When present, use the Format specified inside the object.
3. Use the setting of the variable ColourFormat in the '\_Config' header.
4. Use the default 'Simple' format.

## 6.9 Ruler, RuleOff, Divider, HR

In some situations you may want to separate groups of icons from one another. To do so, a horizontal ruler may be useful. The ruler is just an object descriptor, and will not produce a variable in the *Config* file. It does however control the appearance of the window. At present the descriptor for a ruler is made up of only two variables, e.g.:

```
Type=Ruler
Height=20
```

This will place a horizontal ruler across the window just below the last icon that was created. As a ruler needs less space than the average icon, it may be useful to specify an alternative height. In practice a **Height** value of 20 will be suitable. The result may be something like this:



## 6.10 URL, FTP, WebSite, Web, Link, HTTP, Internet

As the Internet is used more and more, it may be necessary to enter a web address or an e-mail address in a writable icon. E.g. a web browser may want to define the default web page. Additionally it is becoming increasingly popular to include buttons in a program's Info box, to link to a web page or e-mail address.

An example of a descriptor for such a field is given here:

```
{
  Name = SearchEngine
  Text = Search
  Type = URL
  Size = 256
  Help = Put here the address of your favourite search engine
  Default = http://www.google.com
}
```

which will result in the following icon:



As we haven't specified a **width** parameter, the icon is extended the the far right of the window, but enough space is left for a *Connect* button. Clicking the *Connect* button will cause the specified page to be opened in your current browser.

In the above example, the icon points to a web page, as indicated by the **http://** prefix. Instead of specifying a web page, you may also enter a e-mail address, a ftp server, etc. E-mail addresses are recognised by the '@' sign and will be linked to your current e-mail package whenever the *Connect* button is clicked. Other internet protocols are identified by using the correct prefix (e.g.: ftp://).

The above icon will result in a variable in the *Config* file, that may look like this:

```
Search=http://www.google.com
Support=support@xat.nl
Updates=ftp://www.riscos.com/ftp
```

## 6.11 Button

Sometimes it may be useful to add a button to a window, e.g. to allow the user to open a folder. The data type 'Button' allows you to do just that. It will not produce a variable in the 'Config' file.

When creating a Button, the `text` variable is used for the text to be displayed inside the button. The actual action, which can be anything that is normally passed to the CLI, should be defined in the `default` variable. Alternatively, the button can send a message to your application (see below).

The following descriptor:

```
{
  Name = Sample
  Text = Sample
  Type = Menu
  Help = Select a sound sample from the menu
  Menu = #Sample/files;<TouchMe$Dir>.Sound
  Default = Default
  Width = 350
}:
{
  Type = Button
  Text = Find...
  Default = Filer_OpenDir <MyApp$Dir>.Sound
  Help = This will open the Sound-directory
}
```

will produce this:



Clicking the *Find...* button will cause the directory viewer '`<MyApp$Dir>.Sound`' to be opened.

Alternative actions may be implemented by executing, say, an *Obey* file this way. The *Obey* file could in turn start another application, execute a utility, etc. Of course, the \*-command 'Filer\_Run' could also be used for this purpose. Any command specified in the Default-variable is executed by calling Wimp\_StartTask, unless it starts with a '\*' in which case it will be sent straight to the CLI. Some examples:

<code>Default = Filer_OpenDir &lt;MyApp\$Dir&gt;.Sound</code>	Executed by Wimp_StartTask
<code>Default = *Close</code>	Executed by the CLI
<code>Default = #createNewFile</code>	WIMP message sent to your application <sup>1</sup>

In case you want to send a signal up to your application when the user clicks a button, the Default-variable should start with a '#'. It is also possible to combine this function with SystemVariables generated by ConfiX (see chapter 5.1.2 SysVar) in order to include a choice or selection made elsewhere, e.g.:

```
Default = #Launch <myAppVar$Value$Main@FileName>
```

This will generate a WIMP-message<sup>1</sup>, which might look like this:

```
Button Launch myFile
```


### Remarks

1. Starting a line with a '#' allows the button to generate a WIMP-message (see chapter 7.2). The message will start with 'Button' followed the text specified after the '#'. The message and your text are separated by a single space. This function is introduced in version 1.10 of ConfiX.

## 6.12 Icon, Sprite

!ConfiX provides limited support to place sprites inside the window. Sprites may be added to clarify the window's appearance and will not be treated as variables in the *Config* file (just like rulers and buttons). Any sprites added to the window this way, should be present in the '\_Tabs' sprite file in the *Files* directory. Sprites should not be (much) higher than the standard height of a line to avoid a cluttered display.

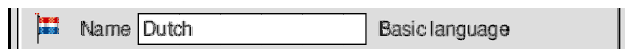
Suppose we want to add the Dutch flag in front of a string icon. First of all we would add the following sprite to the '\_Tabs' file:

 The name of this sprite is 'flag\_nl'

Next we would define some icons using something like this:

```
{
  Type = Sprite
  Default = flag_nl
}:
{
  Name = Name_0
  Text = Name
  Type = String
  Width = 358
  Help = This is the name of the basic language (used in menus)
  Default = Dutch
  Comment = Basic language
}
```

The result would look like this:



### Options

The following options can be specified in the `Options` variable:

#### **-border <type>**

Add a WIMP-type border to the icon. E.g. **-border 2** will produce the same border as for a read-only field (i.e. slabbed-in border). The border can be turned off by setting **-border 0**.

### Remarks

1. In versions of ConfiX prior to 0.86, the `Width` parameter must be specified. If it is *not* specified, a default width of 0 will be used, resulting in a mis-placement of the icon and potential screen-redraw problems. From version 0.86 onwards, the actual width and height of the sprite are used to adjust the values specified in your definition if either of them is too small.
2. From version 0.86 onwards, when the default sprite doesn't exist, a minimum width and height of 48 OS-units will be assumed, unless otherwise specified in your definition.



## Making the sprite dependant on the value of another variable

The `Default` variable can be used for a special case by putting a `#` sign at the start of the value. It allows a different sprite to be selected depending on the value of *another* variable in the same group (e.g. a menu selection). An individual sprite file must be present, containing all sprites can can be displayed in this object. The name of each individual sprite should match the value of the specifiec `<var>`. In this case the syntax is as follows:

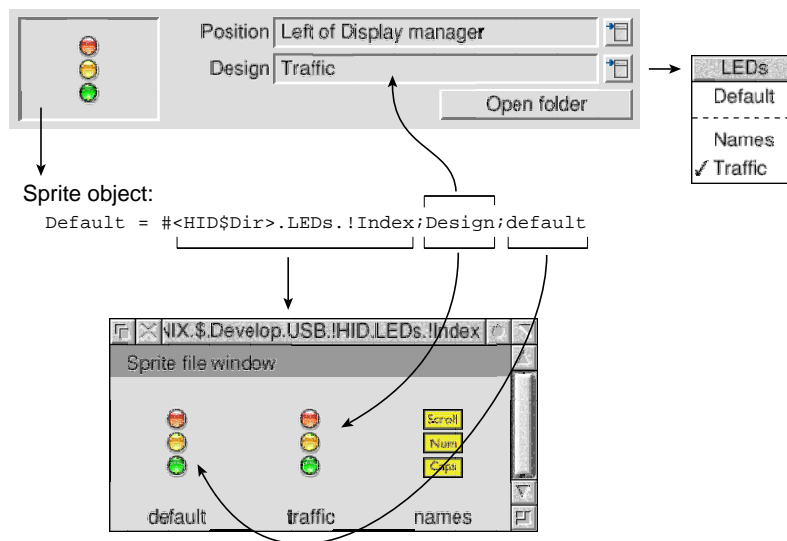
```
Default = #<path>;<var>;<default>
```

**path**            Pathname to a sprite file containing the sprites to be loaded (e.g. `!Index`).  
**var**            Name of the variable (in the same group) that this one is depending on.  
**default**        Name of the default sprite (from the sprite file pointed to by `<path>`).

Example:

```
Default = #<HID$Dir>.LEDs.!Index;Design;default
```

The diagram below shows the relation between the various components:



The object definition for the above example looks like this:

```
{
  Type = Icon
  Default = #<HID$Dir>.LEDs.!Index;Design;default
  Width = 220
  Height = 162
  Options = -border 2
  Help = This design will be used for the keyboard LEDs in the iconbar
  Enable = Enable,1
}-
{
};
{
  Name = Position
  ...
}
```

## 6.13 OLE

When discussing the data type *Button*, we've seen that it is possible to add a button with a CLI action, e.g. to open a directory. Although this will be sufficient in most cases, you may want the user to specify a directory path, or a path to a file. In those cases it is impossible to add a button to open the required directory, as the path is unknown at the time you create the *Button*.

To overcome this problem, !ConfiX can create an OLE icon for you. The OLE icon consists of a writable icon where the user can type the full pathname. Alternatively, the user may drop a file or directory directly into the writable field. !ConfiX will then enter the path for you. To the right of the writable icon, an OLE icon will be added (a button with an eye).. Clicking this button will open the directory viewer or, in the event of a file, will try to load it in your current editor.

An OLE field is created as follows:

```
{
  Name = Path
  Text = Path
  Type = OLE
  Help = This is the path where the logfile will be stored
  Default = <MyApp$Dir>.Log
  Drop = path
}
```

which will produce this:



In this example, clicking the eye-icon to the right of the writable field, will open the directory holding the Log files.

A new variable '*drop*' has been added to control the functionality of the OLE icon. It controls the action when dropping a file or directory folder inside the writable field. **OLE** icons behave, in the same way as **String** icons (described in paragraph 6.2), except that an OLE-icon is added to the right of the writable field. Please refer to paragraph 6.2 for a detailed description of the *Drop* variable.

Clicking the OLE-icon issues a **Filer\_Run** command, with the contents of the writable field as a parameter or path. If the path would lead to a file, it would be opened in the current text editor. If a directory was specified, it would be opened and the contents would be visible in a standard filer window.

### Remarks

1. Added in version 0.82: ConfiX now checks whether the specified directory is actually an application (i.e. the name starts with a '!'). If it does, it checks for the existence of the !Run file inside the application. If it exists, the application will be executed as usual. If it doesn't exist however, it issues a **Filer\_OpenDir** command instead.

## 6.14 IP address and netmask

This field is intended to be used to enter an **IP address** or a **netmask** as per Internet conventions. An IP address always consists of 4 numbers, ranging from 0 to 255, separated by full-stops, e.g.

192.85.162.4

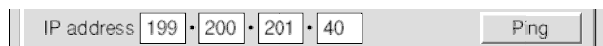
A netmask is entered in a similar way, e.g.

255.255.255.0

An IP field is created as follows:

```
{
  Name = IPAddress
  Text = IP address
  Type = IP
  Width = 350
  Default = 199.200.201.40
  Options = -fix -ping
  Help = Enter the required IP address here
}
```

This will result in a field that looks like this:



Another example (a netmask in this case):

```
{
  Name = Netmask
  Text = Netmask
  Type = IP
  Width = 350
  Default = 255.255.255.0
  Menu = IP Address;Class A,255.0.0.0;Class B,255.255.0.0
  Options = -fix -default -none Default
}
```

Which will result in this:



A menu may be added to supply some useful defaults, by supplying a **Menu** variable as shown in the above example. Furthermore the following switches may be added, using the **Options** variable:

### -fix

If this switch is present, the size of the 4 writable fields is fixed and the (optional) menu-icon will be added to the end of the field. In other words: the field width will not be shrunk to fit the menu-icon. This may be useful when trying to align multiple IP fields, some with and some without a menu.

### -ping

Adds a **Ping** button to the field, allowing pinging of the Address. Clicking the button will result in a TaskWindow to be opened, showing the results of the Ping command.

### -default

Adds a button called **Default** to the field, allowing the default value to be entered (taken from the `Default` variable in the `_Config` file).

### -none <entry>

If this switch is present AND a menu-icon is present, an extra item will be added at the top of the menu, called `<Entry>`. Selecting this item from the menu, will result in the field being cleared. Some applications use this to select the default value.

### -wide

Use one large field, rather than 4 numerical ones, to allow entry of IP names as well as numbers.

## 6.15 BlockDriver

If you want to support the use of a serial port from your application, you may want to set the parameters for the port from your Config file. Creating support for a serial port can be done in two different ways:

1. using BlockDrivers, or
2. using DeviceFS

This field type is used to support the BlockDriver specification as created by Hugo Fiennes and maintained by X-Ample Technology at: <http://www.xat.nl/en/riscos/sw/bd/>.

When configuring a serial port, one needs to set one or more of these parameters:

- |                |                    |
|----------------|--------------------|
| 1. Interface   | (e.g. InternalPC)  |
| 2. Port        | (e.g. 0)           |
| 3. Speed       | (e.g. 115200 baud) |
| 4. Data format | (e.g. 8N1)         |
| 5. Data flow   | (e.g. Hardware)    |

By default, !ConfiX will only allow the user to set **Interface** and **Port** number. The other fields are optional and should be specified in the **Options** variable.

When selecting an **Interface** from the menu, !ConfiX will load the specified driver into its own workspace, to extract some useful information from it. For this to work, the application !SerialDev should be present on your computer, preferably in the !Boot.Resources folder. If it is not found, !ConfiX will generate an error and allow you to download the most recent version from the website.

A typical BlockDriver field can be created like this:

```
{
  Name = Interface
  Text = Interface
  Type = BlockDriver
  Default = InternalPC,0,115200,8N1,hw
  Options = -button -speed
}
```

which will result in this:



In this example, the **-button** switch is specified, in order to get a **Setup** button to the right of the field. If this switch is not present, a menu-icon will be used instead. All parameters used to configure the serial port are visible in the icon, separated by commas.

### Options

The following switches are allowed in the **Options** variable:

#### **-fix**

If this switch is present, the size of the icon is fixed and the menu-icon (or Setup button) will be added to the right. The size of the icon will not be shrunk in order to fit the menu-icon.

#### **-button**

This switch can be used to get a Setup button instead of the default menu-icon.

#### **-speed**

Allow editing of the speed parameter.

#### **-data (or -format)**

Allow editing of the data format (e.g. 8N1).

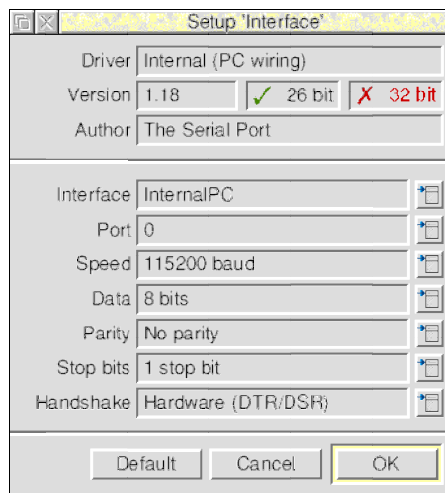
#### **-flow (or -handshake)**

Allow editing of the flow control (handshake).

## **-baud**

Allow entry of user-defined baud rate.

If all switches are present (i.e. `Options = -speed -data -flow`), clicking the **Setup** button (or menu-icon) will pop-up a window to alter the serial settings:



The top part of this window contains general information about the selected block driver. In the above example it shows the name of the driver Internal (PC wiring). Immediately below that, is the version number and two important flags to show compatibility with existing 26-bit versions of RISC OS and the new 32-bit variant. Finally the name of the author is given.

The lower part of the window can be used to enter the required settings. Depending of the switches (in the Options variable) one or more fields may be greyed out. Clickin one of the menu icons at the right of each field, will reveal all possible choices, which may vary between different block drivers.

Klik **OK** to confirm any changes. The BlockDriver field will be updated accordingly. The **Default** button at the bottom left of the window can be used to revert to the default settings as defined in the `_Config` file. Note that the **Default** button will only be present if the main !ConfiX window also has a **Default** button.

When the user selects an Interface from the first menu, the appropriate BlockDriver will be loaded into !ConfiX' own workspace and all other menus will be changed to reflect the limits for that interface. This way the user will never be able to enter an illegal combination of parameters.

The resulting `Config` file will contain a line like this:

```
Interface = InternalPC,0,115200,8N1,hw
```

representing the internal serial port with PC pinout, port 0, 115200 baud, 8 databits, no parity, 1 stopbit and hardware flow control. It is up to your application to interpret this data appropriately and translate it into useful parameters and bitflags for the blockdrivers to use. An empty parameter should be treated as the default value known by your application.

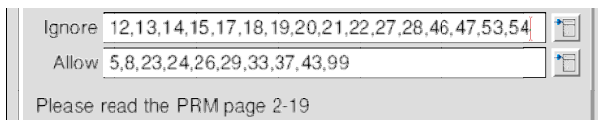
## 6.16 Comment, Remark

This is a passive object in that it doesn't allow any selection to be made. Comments may be used to add general comments and remarks to a TAB, e.g. a warning to the user. Comments are very easy to add and require only a few variables to be entered. However, additional variables can be used to control the overall appearance of the comment.

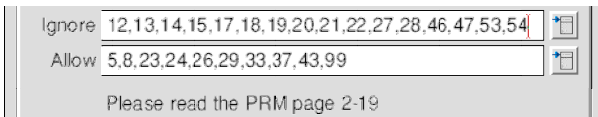
By default a comment has the height of a single line, unless specified otherwise. Single line comments are always left aligned and will have no border (by default). The following definition:

```
{
  Type = Comment
  Text = Please read the PRM page 2-19
}
```

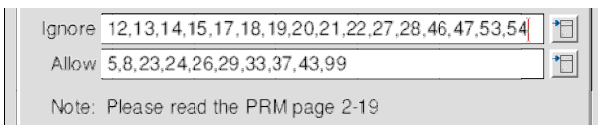
will produce something like this:



If you want to align the text box with the other icons, you may add the `Legend` variable:



Result when specifying:  
`Legend =`



Result when specifying:  
`Legend = Note:`

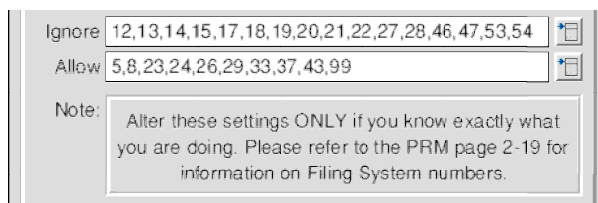
Please note the extra space between the upper two icons and the comment line, which was created by adding a null object in the `_Config` file of which only the `Height` is specified; like this:

```
{
  Height = 8
}
```

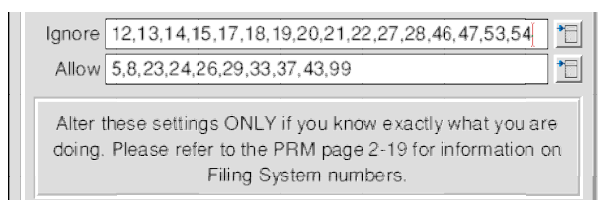
By default a *Comment* box has the same height as any other item, i.e. one line. Larger boxes are possible by specifying an alternative height. If height is specified, a border will be added to the box. Alternative borders may be specified and any of the 16 desktop colours may be used. The following definition:

```
{
  Type = Comment
  Legend = Note:
  Text = Alter these settings ONLY if you know exactly what you are doing. Please refer...
  Height = 158
}
```

will produce this:



In this example the variable:  
`Legend = Note:`  
is used to align the box with the other icons.



Same example, but this time the `Legend` variable has been omitted. As a result, the *Comment* box will be stretched to fill the window.

A number of switches can be used in the `Options` variable to allow further control over the *Comment* box.

### **-col <wimpcolour>**

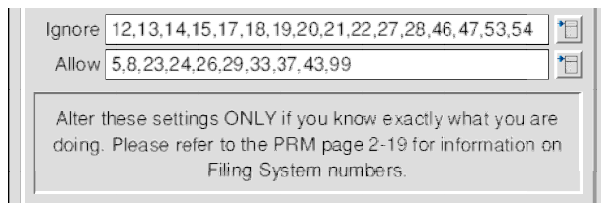
By default the colour of the text inside the *Comment* box will be black (WIMP colour 7). The text colour can be changed by specifying the **-col** switch. E.g. if you want the text to be red, you would add the following to the `Options` variable:

```
Options = -col 11
```

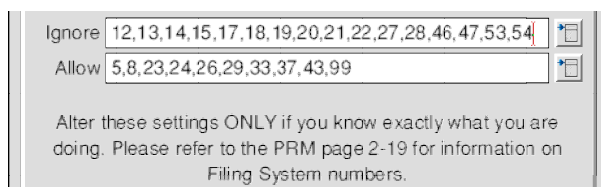
### **-border <type>**

By default, *Comment* boxes that have the default single line height, will have no border. *Comment* boxes of an alternative height (the `Height` variable is specified), will automatically get a border assigned. Border types are the same as for the window manager (WIMP). By default, border 4 is used as shown in the previous examples. Alternative borders may be selected by adding the **-border** switch to the `Options` variable. E.g. a 3-D border may be used by specifying:

```
Options = -border 2
```



Use **-border 0** to turn the border off altogether.



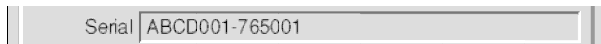
## 6.17 Info

This object is available from version 0.50 onwards and allows a non-editable field to be displayed. It can be used e.g. to display a serial number. If such a number needs to be present as a 'key' for the application, but the user should not be allowed to edit the number, the *Info* object might be useful. *Info* boxes behave much like *Comment* boxes (see previous paragraph), but with the difference that the contents of the field are written into the *Config* file when the user clicks the *Save* button.

The following description:

```
{
  Name = SerNo
  Text = Serial
  Type = Info
}
```

will result in something like this:



The field will automatically extend to the right of the window. If a smaller box is required, the `Width` variable should be used.

### Options

The following switches are allowed in the `Options` variable:

#### **-col <wimpcolour>**

By default the text in the *Info* field will be displayed in black, however you may change it into any of the 16 wimpcolours, by supplying the **-col** switch, followed by the colour number. E.g. the following entry will result in red text to be used:

```
-col 11
```

#### **-border <wimpborder>**

By default the border of an *Info* box will be the same as for any other non-editable field (border 2), but alternative borders may be selected by supplying the **-border** switch, followed by the appropriate wimp border number (as used in icons and templates). To turn the border off completely, one would specify `-border 0`. E.g.:

```
-border 0
-border 1
-border 2 (default)
```

#### **-hide**

If you don't want the *Info* field to be visible to the user, you should add the **-hide** switch. This will result in the *Info* box to be created off-screen, so that it is not visible, but will return the contents of the field in the resulting *Config* file.



## 6.18 Slider, Fader

Sliders are a good and graphical alternative to using Integers. They may be particularly useful when entering colours, sound levels, etc. Sliders (also called 'faders') are available from version 0.51 onwards and can be displayed in a variety of ways. The simplest form of a slider can be created by this definition:

```
{
  Name = Volume
  Text = Volume
  Type = Slider
  Min = 0
  Max = 255
  Default = 200
  Help = Use this slider to set the volume
}
```

In this case we want to create a slider that ranges from 0-255 in steps of 1. The default value is set to 200. The result will be:



The length of the slider will automatically adjust to the width of the window and !ConfiX will take care of all calculations with respect to the required values. By default the slider will be displayed in dark grey, inside frame with border 2 (slabbed in). Different colours, borders, etc. may however be selected by supplying one or more of the switches below in the `Options` variable.

### **-fix**

By default the slider will adjust itself to fit the screen. If this is not allowed, you may specify the `width` variable. However, whenever extra icons are added (editable value and arrows) the `width` will be adjusted accordingly. If this is not required, a fixed `width` may be forced by supplying the **-fix** switch.

### **-border <wimpborder>**

The switch `-border` can be used to select any of the available WIMP borders (as used in icons and templates). By default wimp border **2** is used (slabbed in), but alternative ones may be selected. If no border is required, you should specify **-border 0**. The slider would then look like this:



### **-value**

By default, the slider will be the only way to enter a value in this field. Optionally, an editable field may be added (with accompanying up/down arrows) to allow a more flexible data entry. To do this, the **-value** switch should be supplied. It is advised to include the `Size` variable as well, so that !ConfiX can calculate a sensible size for the editable field. Please note that `Size` does not refer to the width of the field in OS-units, but the number of characters instead. For the above example, one would enter `Size = 3`. The result could be something like this:



### **-scale <n>**

This switch can be used to add a visual aspect to the slider. The `-scale` switch should be followed by a number, representing the step size (in your own units). In the above example, we've defined a slider ranging from 0-255 (256 units) and we may want to add a scale with a step size of 16 units. We would add the following to the `Options` variable: `-scale 16`, and the result would look like this:



The scale will be displayed in mid-grey and a small arrow indicates the default value (as defined by the `Default` variable).

### **-leftright**

When using an editable field, the up and down arrows are supplied by default. These may be replaced by left and right arrows by adding the switch `-leftright`. This will also work when `-value` is not supplied.

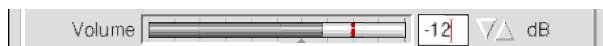


### **-mark <n>**

In addition to the marker above, another marker may be added to indicate a reference point (e.g. the 0dB point of a sound slider). Add the **-mark** switch followed by the required value (in your own units). The marker will be shown as a red band inside the slider. Consider the following definition:

```
{
  Name = Volume
  Text = Volume
  Type = Slider
  Min = -80
  Max = 20
  Default = -20
  Help = Use this slider to set the volume
  Size = 3
  Options = -value -scale 10 -mark 0
  Comment = dB
}
```

This will produce a sound slider ranging from -60 to +20 dB, with a marker at 0dB. As the default is set to -20, a small arrow will be shown at that point as well. The result will be something like this:



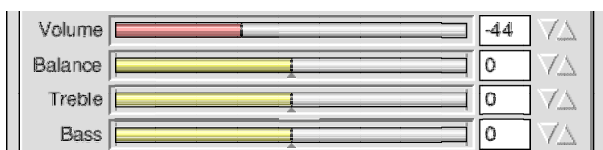
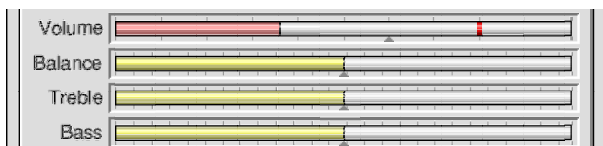
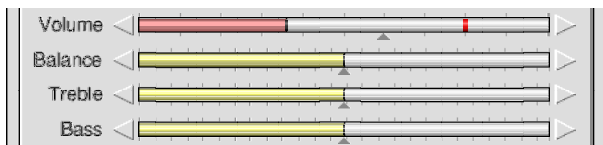
### **-col <colour\_name>**

By default the colour of the slider will be dark-grey. Alternative colours are available by specifying the the switch **-col** followed by the colour name. The following colours/names are available:

**red, green, blue, aqua, magenta, yellow, brown, cyan, cream, mint, pink and orange**

### **Some examples**

Below are some examples of sets of sliders, each with different settings in order to achieve a different graphical result.



## 6.19 Spacer

---

This object type allows vertical whitespace to be inserted between two successive fields. The whitespace is specified in OS units

t.b.a.

## 6.20 Caption

---

Title

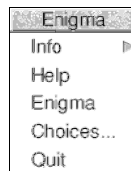
t.b.a.



## 7 Calling !ConfiX from your application

!ConfiX can be used by many applications at the same time. Each time an application wants to open a choices window, a new instantiation of !ConfiX will be started. For this to work correctly, !ConfiX should be seen by the filer, before it is called by your program. The best place to put !ConfiX is therefore in the Resources directory of the main harddisc (in the same directory as the !System folder). When the system is started, !ConfiX will setup the system variable <Confix\$Dir> to point to its position.

When altering the configuration of a program, the user would generally select the **Choices...** option from the application's iconbar menu, e.g.:



Your application should first check the system variable <Confix\$Dir>. If it doesn't exist, it should launch the **Config** file in the current text editor (e.g. !Edit), by issuing a **Filer\_Run** command. If the system variable <Confix\$Dir> does exist, The !ConfiX application may be launched using the SWI "Wimp\_StartTask". In the command tail, you may pass some additional options and values as shown in the example below.

The following example shows a procedure in BBC BASIC that can be used by your program to launch !ConfiX.

```
DEFPROCconfix(file$,tab%)
  LOCAL temp$
  temp$ = FNread_var_val("Confix$Dir")
  IF temp$ = "" THEN
    SYS "XOS_CLI","Filer_Run "+file$
  ELSE
    SYS "Hourglass_On"
    temp$ = " -res "+appdir$+".Files"
    temp$ += " -file "+file$
    temp$ += " -mes MyAppRes:Messages"
    temp$ += " -task "+STR$(taskid%)
    temp$ += " -tab "+STR$(tab%)
    temp$ += " -pos center"
    SYS "Wimp_StartTask", "<Confix$Dir>"+temp$ TO ConfixHandle%
  ENDIF
ENDPROC
```

The procedure is generally called from the section in your application that decodes the iconbar menu selection. On entry to the procedure you supply the path name to the **Config** file e.g.:

```
<MyApp$Dir>.Files.Config
<Choices$Write>.MyApp.Settings
```

The parameter **tab%** can be used to select a default TAB when the main configuration window is opened. If you don't want to select a default TAB, you may use 0 (zero) for this parameter.

In the above example the local variable **temp\$** is used to build the command string for the SWI "Wimp\_StartTask". A number of parameters may be passed this way, some of which are optional. The following section describes each option in more detail.

### Notes:

1. Please note that the line containing the **-file** option (printed in **bold** in the above example) is optional. If it is omitted, !ConfiX will look for a file called *Config* inside the folder specified by the **-res** option.
2. Note that the *TaskHandle* of the ConfiX task that has been started, is returned by the SWI 'Wimp\_StartTask' (it is called ConfixHandle% here). Use this handle when sending commands to ConfiX (see chapter 7.3).

## 7.1 Passing parameters to !ConfiX

---

### **-res <path>**

Pointer to your application's resources

When launching !ConfiX, you should pass a pointer to the resources that are used to build the configuration window. If you've set your application up as described in the previous chapters, you would point to your application's **Files** directory. Some examples:

```
-res MyApp:Files
-res <MyApp$Dir>.Files
```

### **-mes <path>**

Pointer to the Message file

To allow internationalisation of your application, RISC OS makes extensive use of the *MessageTrans* module. Generally, an application uses a file (often called **Messages**) to translate the application's tokens into readable text. Various *Messages* files can be supplied for foreign versions of your software. In the example we've used the following line:

```
-mes MyAppRes:Messages
```

This will point to the correct *Messages* file for the current language. If internationalisation of your software is not required, you may point directly to your *Messages* file, e.g.

```
-mes <MyApp$Dir>.Resources.Messages
```

However, making your application foreign-language-friendly is not a major task and can easily be incorporated using the **ResFind** utility. Details of ResFind can be found on the Internet, but if you fail to obtain a copy, please drop us an email at [support@xat.nl](mailto:support@xat.nl).

If you supply a *Messages* file, you may supply foreign translations for all text in the configuration windows. If a translation fails, the default text from the *\_Config* file will be used instead. Please refer to the chapter Internationalisation for a detailed description of the translation mechanism.

### **-task <handle|name>**

Pointer to your task handle

In order to allow !ConfiX to pass messages back to your application, you should supply it the current handle of your task. The task handle is the number returned from the SWI "Wimp\_Initialise" when the task is started. In BASIC this line would look something like this:

```
SYS "Wimp_Initialise",310,!task%,AppName$,task_messages% TO ,mytask%
```

The returned variable `mytask%` should be passed in ASCII format to !ConfiX, hence the BASIC line:

```
temp$ += " -task "+STR$(mytask%)
```

Please note that the *tasknumber* is supplied rather than the *taskname*, as !ConfiX should be able to identify your application when you have several instances of the same program running simultaneously. When calling ConfiX from the command-line or from, say, an Obey-file, you may specify the *TaskName* instead, although this has the limitation that this method cannot be used reliably when using more than one copy of your application simultaneously.

### **-tab <tab>**

Selecting a default TAB

If you have more than one configuration window for your application, each of them will be represented by a TAB. Initially, when the configuration window is first opened, the first TAB will be selected. Alternatively, you may supply the optional **-tab** switch that allows a particular TAB to be selected immediately when the window opens. This may be particularly useful when *Choices...* is selected from a specific part or window inside your application (e.g. a *SetUp* button in a Print dialogue box).

### **-file <filename>**

#### Pointer to the Config file

This parameter is optional and should only be supplied if your *Config* file has a different name or is stored in a different location. By default !ConfiX assumes the file to be called **Config** and to be present inside the **Files** directory of your application (as pointed to by the **-res** switch).

In certain situations however, you may decide to put the configuration files of several applications in a common directory (e.g. the new RISC OS *!Boot.Choices* directory). If such is the case, a pointer to the actual configuration file should be supplied, e.g.:

```
-file <Choices$Write>.MyApp.Settings
```

This path will be used by ConfiX to read and write your current choices. If the file does not exist when opening the ConfiX window, the default settings will be used instead.

*Please note that rather than specifying this parameter, you may also use the 'File' and 'Save' variables in the header of the '\_Config'-file. See chapter 5.1 for more information.*

### **-load <filename>**

#### Pointer to the Config file to load

This parameter is optional. In some situations it may be desirable to specify an alternative path for loading your Config file. In such cases you may specify the -load parameter, e.g.:

```
-load Choices:MyApp.Settings
```

Although this looks similar to the previous parameter, it allows a path to be specified rather than a file. In the example above, ConfiX will try to find the given file in all directories specified in the 'Choices:' path. This is the recommended method for loading and saving choices files.

*Please note that rather than specifying this parameter, you may also use the 'Load' variable in the header of the '\_Config'-file.*

### **-pos [<x>,<y>]**

#### Defining the initial window position

This parameter can be used to determine the initial position of the configuration window. If the parameter is omitted, the window will always be opened in the bottom left corner of the screen, leaving the iconbar free. The option **-pos** may be followed by none or two parameters. If no parameter is supplied (or only one parameter) the window will always be centered on the screen. If two parameters are supplied they will be used as the **x** and **y** co-ordinates respectively. Some examples:

```
-pos 0,0  
-pos 100,200
```

Rather than supplying direct co-ordinates as in the above example, you may use an expression to determine the position. Legal expressions for the x-coordinate are:

```
Centre (or Center, or Middle)  
left  
Right
```

The y-coordinate recognises the following expressions:

```
Centre, Center, Middle  
Bottom  
Top  
Iconbar
```

The difference between *Bottom* and *Iconbar* is that the latter will leave the iconbar free. Some examples:

```
-pos centre,centre  
-pos left,top  
-pos right,iconbar
```

### **-default 0|1**

#### Hiding/showing the Default button

By default, !ConfiX will show a *Default* button in the configuration window. Clicking this button will set all variables to their initial settings as defined in the *\_Config* file. You may force the *Default* button to be invisible or visible, using the following switches respectively:

```
-default 0
-default 1
```

### **-profiles <n>**

#### Hiding/showing the Profiles button

By default the configuration window will show a Profiles button. Clicking this button will present a menu of files that are present in the **\_Profiles** directory at the same level as the 'Config' file. A value of 0 turns the profiles mechanism off, whilst any positive value allows you to select a suitable profiles system. Please note that this setting will override the setting in the '*\_Config*' file header. The following settings of **profiles** are possible (see chapter 5.1.2 for further details):

- 0 Off
- 1 Use standard 'Profiles'-button
- 2 Use a menu icon rather than a button
- 3 Add profiles to the main ConfiX-menu, rather than adding a button
- 4 ConfiX behaves as Profile Manager
- 5 Button launches Profile Manager

### **-help 0|1**

#### Forcing the Help-line on or off

Whenever possible, the switch **-help** should be omitted, leaving the user in control over the use of the *Help*-line. The *Help*-line may be turned *on* or *off*, at the user's convenience. There may be situations however, where you want to force the *Help*-line to be turned *on* or *off*, regardless the user's preferences. To do so, use one of the following switches:

```
-help 0
-help 1
```

### **-hide <tab>[,<tab>,<tab>,...]**

#### Hiding TABs from the user

For some applications it might be useful to hide a particular TAB, or group of TABs from the user. This may be the case, for example, when the user doesn't have the right access privileges for a certain operation. The switch **-hide** may be used to hide one or more TABs from the user, e.g.

```
-hide 2
-hide 2,5,8
```

The TABs will still be visible, but their icons will be greyed out and the TAB will not be selectable by the user.

### **-ok <filename>**

#### Adding an OK button to the window

By default the configuration window will not feature an OK button. Instead the user has the choice between a *Save* button and a *Cancel* button. To confirm the new settings, the user would generally click the *Save* button, but the disadvantage would be that it makes the changes permanent (as they are written back to the *Config* file). If it is required by your application, an OK button may be added to the ConfiX window, which enables the user to set alternative choices for this session only. Once the program is restarted, the default *Config* settings will be used instead. An OK button may be added by adding the **-ok** parameter, followed by a path to the alternative file, to the command line, e.g.:

```
-ok <MyApp$Dir>.Files.OK
-ok OK
-ok <Choices$Write>.MyApp.OK
```

Whenever the user clicks OK, the new settings will be written to the **OK** file (as specified above), rather than to the standard *Config* file. If you only specify the leafname (as in the second example) the file will be stored in



the path specified by the **-res** parameter. In addition ConfiX will send the message 'ConfigOK' to your application. The message 'ConfigOK' will be followed by the full pathname to the OK-file.

It is advised that your program deletes the OK file when starting program, so that it won't confuse your application. Reading and deleting the OK file is the responsibility of your application.

### **-layout <layout>**

#### Selecting an alternative layout

By default the user is in control of the appearance of the configuration windows. Initially, ConfiX was supplied with a single layout, consisting of a large window with a row of graphical TABs at the top, but in version 0.54, a number of alternative layouts have been introduced. Later versions may contain new additional layouts.

The user can select a preferred layout to be used for all configurations handled by ConfiX. Some application authors however, may wish to force ConfiX to use a given layout, e.g. to match the user manual for that application, or simply to be consistent with other windows used in the application. There are two ways to achieve this:

1. by supplying the `Layout` variable in the `'_Config'` header, or
2. by supplying the command-line switch **-layout** followed by the name of the required layout. The latter takes preference over the variable. Please refer to paragraph 5.1.4 for a list of available layouts.

### **-groupicon <0|1>**

#### Showing/Hiding the group icon

ConfiX uses icons to identify groups of settings in some layouts. Certain layouts however, identify the groups by text or buttons rather than icons. In such cases, the group icon may be placed somewhere in the main window. By default this feature is enabled and the user is in control of it (via ConfiX Choices), however there are two ways to override this:

1. by supplying the `GroupIcon` variable in the `'(Config'` header (paragraph 5.1.2), or
2. by supplying the command-line switch **-groupicon <0|1>**. The latter takes preference over the variable in the header.

The value that follows the **-groupicon** switch has the following meaning:

- |   |   |
|---|---|
| 0 | Never show the group icon                     |
| 1 | Always show the group icon (when appropriate) |

### **-name <name>**

#### Specify alternative name

By default, the client's name is retrieved from the TaskID, passed to ConfiX via the command line (**-task**). However, an alternative name may be specified, in case your application needs to create multiple ConfiX windows, each with its own title. This name may also be used in the window title, by adding a variable to the Title specified in the `_Config` header, e.g.:

Header variable:	<code>Title = %0 choices</code>
Command line option:	<code>-name Violet</code>
Result:	<b>Violet choices</b>

If no alternative name is specified, the client's name will be used instead.

### **-id <name>**

#### Specify ID

By default, ConfiX returns a WIMP message 'Config' when you click the Save-button. If, however, you use ConfiX for more than one purpose within the same application, it might be useful to have an ID included with the message. Whenever an ID is specified when calling ConfiX, this ID will be added to the Config-message. E.g.:

```
-id mySettings
```

ConfiX will then send the message 'Config mySettings' when the Save-button is clicked. Similarly, if the Cancel-button is clicked, the message 'Cancel mySettings' will be sent (version 1.10 onwards).

## 7.2 Messages from !ConfiX to your application

---

As !ConfiX was launched by your application, you may want it to return a message to your application as and when something significant has happened. !ConfiX will issue a WIMP-message if (1) the *Cancel* button is pressed, or (2) the *Save* button is pressed.

In order to make this system of messages as universal as possible, WIMP-message 0x40D50 has been defined. This allows textual commands to be passed between applications, making it very versatile. The message is sent as a User\_Message (17) and the format of the user message block in R1 is:

```
+0    256
+12   0
+16   &40D50
+20   message string
```

The message string at offset +20 is a typical textual string that contains textual commands. Your program should respond to those commands appropriately or, if you don't want to support the messages, ignore them completely. Commands sent this way, should not be case-sensitive (hence the use of FNupcase in the example below). A typical BASIC example of the command decoder might look like this:

```
DEFPROCpoll(nudge%)
  SYS "Wimp_Poll",0,mes TO r%
  CASE r% OF
    ...
    ...
    WHEN 17,18,19
      PROCrec_mes(r%)
  ENDCASE
ENDPROC
...
DEFPROCrec_mes(action%)
  CASE mes!16 OF
    WHEN &40D50: PROCexecute_message($(mes+20),mes!4)
  ENDCASE
ENDPROC
...
DEFPROCexecute_message(mes$,from%)
  LOCAL up$,pos%,leaf$
  pos% = INSTR(mes$," ")
  IF pos% = 0 THEN
    leaf$ = ""
  ELSE
    leaf$ = MID$(mes$,pos%+1)
    mes$ = LEFT$(mes$,pos%-1)
  ENDIF
  up$ = FNupcase(mes$)
  CASE up$ OF
    WHEN "GETCARET"
      PROCset_caret
    WHEN "CONFIG"
      PROCload_defaults("")
    WHEN "CONFIGOK"
      PROCload_defaults(leaf$)
  ENDCASE
ENDPROC
```

Please note that from version 1.10 onwards, some WIMP-messages may contain an ID. This is only the case if you specified the `-id` switch when calling ConfiX. The message and the ID will be separated by a single space. Also from version 1.10 onwards, ConfiX sends a message 'ConfigCancel' whenever the user clicks the cancel button. It allows your program to respond to it appropriately.

The following messages may be sent by !ConfiX:

#### **GetCaret**

This message is sent by !ConfiX when the *Cancel* button is clicked by the user. The configuration window will be closed and !ConfiX will kill itself. Just before quitting, !ConfiX will send the message "GetCaret" to your application, so that you may regain the input focus (i.e. claim the caret).

#### **Config [<id>]**

Whenever the user has altered some settings, and clicks the *Save* button to confirm, !ConfiX will send the message "Config" to your application and then quit itself. Your application should respond to this message by reloading the *Config* file and setting the internal variables of your program appropriately. If an (optional) ID is passed when calling ConfiX (see chapter 7.1) this ID will be added to the message (version 1.10 onwards). If the ID was 'mySettings', the message from ConfiX will be:

`Config mySettings`

The message will be sent to your application whenever the user:

- Clicks the Save-button
- Hits the F3-key

#### **ConfigOK <filename>**

This message will be issued by !ConfiX whenever the user has clicked the (optional) OK button. Please note that this message also contains the full pathname to the **OK** file used to store the temporary settings for this session. Please use this pathname rather than your own copy of it, just in case !ConfiX finds it necessary to alter it. Please note that the maintenance of the OK file is entirely up to your application (i.e. you are responsible for deleting it once it is no longer needed).

#### **ConfigCancel [<id>]**

This command has been introduced in version 1.10. If an ID is specified (see chapter 7.1) it will be added to the message, separated by a single space. The message will be sent to your application whenever the user:

- Clicks the Cancel-button
- Closes the main window
- Hits the Escape-key

#### **Button <text>**

This message has been introduced in version 1.10. It allows WIMP-messages to be sent to your application when clicking a button in a ConfiX window. Details on how to define Buttons can be found in chapter 6.11. A good suggestion is to use Buttons in combination with System-variables (see chapter 5.1.2). Consider this button-definition, for example:

```
{
  Type = Button
  Default = #doSomething
  Width = 160
}
```

When the user click the button, ConfiX will send the following message to your application

**Button doSomething**

## 7.3 Messages from your application to !ConfiX

---

The protocol described in the previous chapter, can also be used by your application to send messages to !ConfiX. When sending a message, you need to know the *TaskHandle* of your !ConfiX task, as there may be more than one open ConfiX window at any given time. The *TaskHandle* is returned from the SWI 'Wimp\_StartTask' when your first launched !ConfiX (see the example at the beginning of chapter 7).

ConfiX will respond to the following messages:

### **Quit**

Close the ConfiX window and quit. Although not very useful, you may send this message to ConfiX to force it to close down. Please note that ConfiX automatically quits itself when your application is terminated.

### **Config**

Force ConfiX to reload its own choices file. Generally, this command will only be used by ConfiX itself when changing its own choices.

### **Front**

Bring the ConfiX window to the front. This can be useful when the window is hidden behind other windows or iconised. Rather than opening another ConfiX window when the user selects 'Choices...' you may send a 'Front' message.

## 7.4 Loading the Config file

When your program is first started it will need to load the variables from the *Config* file. Furthermore, when the user has altered the contents of the *Config* file, and has confirmed the changes by clicking the *Save* button, your application should reload the *Config* file and act appropriately. The example below may be used as a starting point when loading the *Config* file.

```
DEFPROCload_config
    LOCAL file%,mes$,pos%,var$,val$,group$,sub%
    :
    REM --- setup initial variables here ---
    :
    group$ = " "
    file$ = OPENIN("<CableNews$dir>.Files.Config")
    REPEAT
        line$ = GET$#file%
        CASE LEFT$(line$,1) OF
            WHEN "|" , "#" : REM --- remark/ignore ---
            WHEN "["
                group$ = FNupcase(MID$(line$,2,LEN(line$)-2))
                sub% = 0
                pos% = INSTR(group$,"_")
                IF pos% THEN
                    sub% = VAL(MID$(group$,pos%+1))
                    group$ = LEFT$(group$,pos%-1)
                ENDIF
            OTHERWISE
                pos% = INSTR(line$,"=")
                var$ = LEFT$(line$,pos%-1)
                val$ = RIGHT$(line$,LEN(line$)-pos%)
                pos% = INSTR(var$,"_")
                IF pos% THEN
                    sub% = VAL(MID$(var$,pos%+1))
                    var$ = LEFT$(var$,pos%-1)
                ENDIF
            :
            REM ---strip spaces ---
            :
            WHILE RIGHT$(var$,1) = " "
                var$ = LEFT$(var$,LEN(var$)-1)
            ENDWHILE
            WHILE LEFT$(val$,1) = " "
                val$ = RIGHT$(val$,LEN(val$)-1)
            ENDWHILE
            :
            REM --- force to upper case ---
            :
            var$ = FNupcase(var$)
            :
            CASE group$ OF
                WHEN "PRINTING"
                    CASE var$ OF
                        WHEN "LEFTMARGIN" : LeftMargin% = VAL(val$)
                        WHEN "RIGHTMARGIN" : Rightmargin% = VAL(val$)
                        WHEN "COPIES" : Copies% = VAL(val$)
                        WHEN "PAGENUMBERS" : PageNumbers% = VAL(val$)
                        WHEN "COLOUR" : PageColour% = FNread_rgb_colour(val$)
                    ENDCASE
                :
                REM --- decode other groups and variables here ---
            :
            ENDCASE
        ENDCASE
    UNTIL EOF#file%
    CLOSE#file%
    :
    REM --- initialise certain parts of your application ---
    :
ENDPROC
```

The full source is supplied with the examples together with procedures to decode colours, read hexadecimal values, launch !ConfiX, etc.



## 8 Internationalising your software

Internationalisation is an area that is quite often forgotten by most programmers. However, as RISC OS is also used outside of the UK, you may want to put some work into making your application language-independent. As most applications already use *MessageTrans* to translate the tokens used by your program into useful text, your program may be made language-independent with a minimum effort.

In most programs, the file that holds the translations is called *Messages* and will exist either in the application's directory or inside a further sub-directory, such as *Resources*. Different versions of your *Messages* file may be used to support alternative languages and you may also supply a set of templates adapted for that language. Generally it will prove to be more useful to have a single *Templates* file and put **all** translations in the *Messages* file.

A program that may be useful when internationalising your application is **ResFind**. This small utility, written in BASIC, can be placed inside your application's directory and may be executed from the application's *!Run* file. When executed, it will setup a unique system variable for your application that points to the correct *Messages* file. If it fails to find the configured language, it will use a sensible default (English). In addition to ResFind, there may be other methods and mechanisms to do a similar job.

Once your application has been internationalised, it may be useful to add translations for use by !ConfiX. The default language (English) is already supported, as it uses the text descriptions supplied in the *\_Config* file you've created. ConfiX supports a number of different mechanisms to allow programs to be internationalised easily. This chapter describes each method in detail. The first method uses your application's *Messages* file for most of the translations. This method is now deprecated. Although it's still supported by !ConfiX, the second method should be used in preference. The latter method is described in paragraph 8.2.

### 8.1 Using your application's Messages file

Translations of text from the *\_Config* file into another language, may be carried out by your own system of *Messages* files. You simply add the extra translations to your existing *Messages* file, to which !ConfiX has a pointer (supplied by you when you launch !ConfiX). Translations may be supplied for the **Tab**, **Text** and **Help** variables (described in the *\_Config* file). **Please note that this method is now deprecated. You should use the method described in paragraph 8.2 in preference to this one.**

First you need to know how to construct the tokens necessary to make the translation. Tokens used by !ConfiX always start with a '\_' sign (underscore). Next follows the variable name (Tab, Text or Help). Please note that MessageTrans is case-sensitive, so care should be taken when setting up the files. The following syntax should be respected:

```
_Tab<tabname>:<text>
_Text<tabname><varname>:<text>
_Help<tabname><varname>:<text>
```

Take for example this variable descriptor:

```
[Main]
...
{
  Name=AutoOpen
  Text=Open window
  Type=Option
  ...
}
```

The translations in the *Messages* file would be something like this:

```
_TabMain:Main choices
_TextMainAutoOpen:Open window
_HelpMainAutoOpen:Open this window automatically
```

Keep variable names short to avoid lengthy *Messages* files.



## 8.2 Using different `_Config` files

The easiest and most flexible manner to support multiple languages, is to supply a different `_Config` file for each language. From version 0.73 of !ConfiX onwards, such additional `_Config` files will be recognised. As this method allows for all items to be translated (including menus, etc.) it should be used in preference to the other method described in paragraph 8.1.

The primary '`_Config`' file should be stored inside your *Files* directory just as before. This way, the software can use it as a fallback in case a translation for the current language is missing. For each language you want to support, you should create a different `_Config` file. You may store these additional `_Config` files either in a sub-directory inside *Files*, or in the same directory as your *Messages* file(s).

ConfiX will look for the `_Config` file in this order:

```
<App$Dir>.Resources.France._Config
<App$Dir>.Resources.UK._Config
<App$Dir>.Resources._Config
<App$Dir>.Files.France._Config
<App$Dir>.Files._Config
<App$Dir>._Config
```

The names used here for the various directories, are provided just as an example. The actual names may, of course, be different. As you would normally supply a pointer to the *Messages* file, using the command line option **-mes**, the software knows where to look for the `_Config` file. The *Files* directory is normally pointed to by the **-res** option, so ConfiX may scan that directory too.

### Simple applications

When writing applications that have only one place where *Choices* can be altered, it would be recommended to place the translated `_Config` files inside the *Resources* directory as shown in the first example above. All translated files will then reside in the same sub-directory, which is convenient from a maintenance point of view.

### Complex applications

If your application is more complex than this, and it has several places where choices can be altered, you will generally have created alternatives to the default *Files* directory. Inside the *Files* directory, create a further sub-directory for each language, inside which you put the translated `_Config` file.

### Remarks:

1. Please note that it is advised to put the `_Config` file for the default language (UK) always inside the *Files* directory rather than in the *Resources* directory, to avoid conflicts with earlier versions of !ConfiX.
2. You may want to examine the !ConfiX application itself, to see how it is done there.