Sourcery

Version 1.12 (29-Jul-2012)

Contents

Introduction

Sourcery is intended to be an easy to use development tool that replaces *make* to enable RISC OS applications, modules and libraries to be built quickly and efficiently. *Sourcery* removes the tedium of maintaining the source code and resource files that make up a project and allows the developer to concentrate on the actual development of their project.

*Sourcer*y does not try to impose any restrictions on the tools or compilers that are used to build a project, it simply tries to make the use of these tools as pain free as possible.

Assumptions

This manual assumes that the reader is familiar with application development on RISC OS and also with the C programming language. This is not to say that C has to be used, simply that some of the examples given use C.

Requirements

Sourcery has been tested on RISC OS 4.02 and 4.39. It has been built to be 26/32 bit neutral and does not require the 32 bit C library to be installed. It does however require the latest set of Toolbox modules which can be downloaded from RISC OS Ltd's Web site at http://www.riscos.com.

Getting started

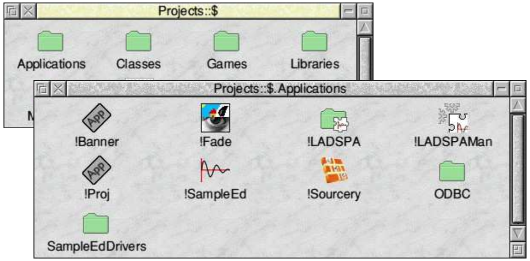
Loading Sourcery

Sourcery is loaded by double clicking on the *Sourcer*y icon in the filer display:

Once loaded an the *Sourcery* icon will appear on the iconbar in the standard way.

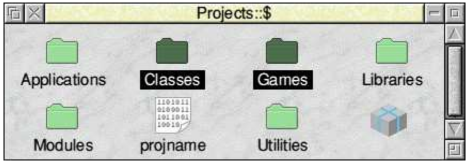
The Sourcery user interface

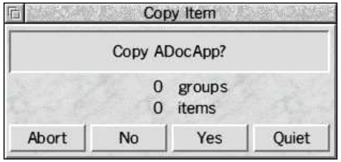
Clicking **Select** on the iconbar icon will display a window containing the projects that are currently defined within *Sourcery*. The first time *Sourcery* is run this will be empty but it can contain projects or project groups. An example is shown below:



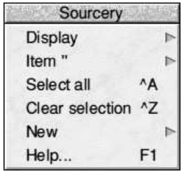
Sourcery has been designed to be easy to use and immediately familiar to anyone used to RISC OS. It therefore uses a Filer like display for many of its windows. This presents the user with objects and directories which can be double clicked on to open new windows.

These objects can be dragged to other windows of the same type either singularly or in groups. A dialog box similar to the Filer Action dialog box is used to confirm the action.



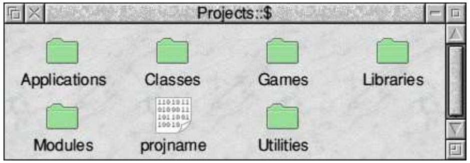


The Filer like display used by many of the windows in *Sourcery* provides a context sensitive menu. The menu is displayed by clicking **Menu** over the window:

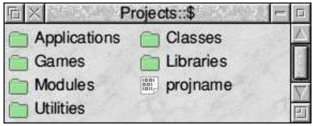


The standard menu has the following entries:

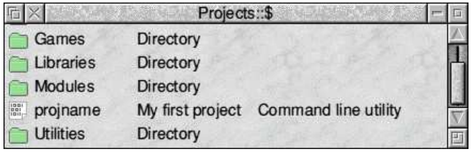
Display
Allows the way the items are displayed to be controlled. Three different types of view are available:



Items are displayed using large icons



Items are displayed using small icons

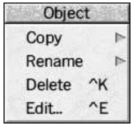


Items are displayed with additional information

Additionally some windows allow the ordering of the items to be specified as well.

Item

The *Item* option leads to a sub menu giving further functionality:



Copy
Allows a copy of the selected item to be made with a new name.

Rename
Allows the selected item to be renamed.

Delete
Prompts the user to confirm the selected items should be deleted.

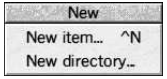
Edit...
Opens an item specific editing window.

Note that some Item sub menus will have additional entries on them which are covered later.

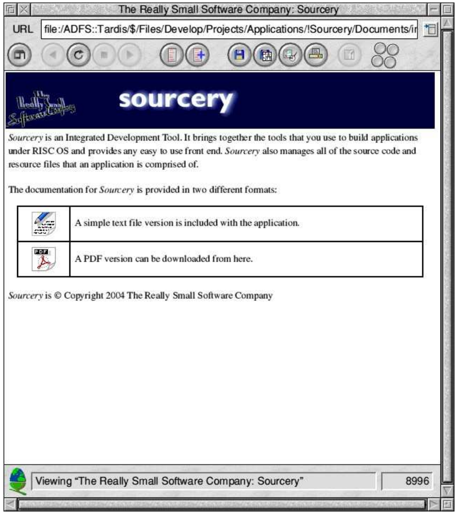
Select all
Selecting this causes all of the items in the window to be highlighted.

Clear selection
This will unselect any highlighted entries in the window.

New
The *New* entry allows a new item to be created. It offers a sub menu which also allows a new directory to be created:



Help
Choosing this item will display the help supplied with the application. If a Web browser is available then that will be used, otherwise a simple textfile will be displayed:


The Really Small Software Company: Sourcery
URL
file:/ADFS::Tardis/$/Files/Develop/Projects/Applications/!Sourcery/Documents/in
sourcery

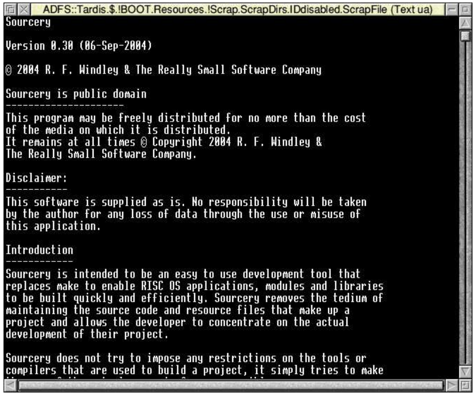
Sourcery is an Integrated Development Tool. It brings together the tools that you use to build applications under RISC OS and provides any easy to use front end. *Sourcery* also manages all of the source code and resource files that an application is comprised of.

The documentation for *Sourcery* is provided in two different formats:

	A simple text file version is included with the application.
	A PDF version can be downloaded from here.

Sourcery is © Copyright 2004 The Really Small Software Company

Viewing "The Really Small Software Company: Sourcery" 8996


ADFS::Tardis.$.!BOOT.Resources.!Scrap.ScrapDirs.IDdisabled.ScrapFile (Text ua)

```
Sourcery

Version 0.30 (06-Sep-2004)

© 2004 R. F. Windley & The Really Small Software Company

Sourcery is public domain
--------------------
This program may be freely distributed for no more than the cost
of the media on which it is distributed.
It remains at all times © Copyright 2004 R. F. Windley &
The Really Small Software Company.

Disclaimer:
-----------
This software is supplied as is. No responsibility will be taken
by the author for any loss of data through the use or misuse of
this application.

Introduction
------------
Sourcery is intended to be an easy to use development tool that
replaces make to enable RISC OS applications, modules and libraries
to be built quickly and efficiently. Sourcery removes the tedium of
maintaining the source code and resource files that make up a
project and allows the developer to concentrate on the actual
development of their project.

Sourcery does not try to impose any restrictions on the tools or
compilers that are used to build a project, it simply tries to make
```

CVS and Subversion

Sourcery has limited support for CVS and Subversion. This is implemented due to the ability to define new commands that can appear on the main menu at project and source code level.

Support is limited to checking code in and out, looking at differences and checking the state of the project. For any other operation the relevant GUI or command line tools must be used.

This manual assumes the user knows how to set up and maintain a CVS or Subversion installation, it is not intended to guide the user in the use of these tools.

CVS can be downloaded from:

http://www.bass-software.com/cvs/

A Subversion client can be found at:

http://www.cp15.org/versioncontrol/

Tutorial

Creating a project

This tutorial explains how to create a simple project, add code to it and compile and run it.

Assumptions

This tutorial assumes you have the following installed on your computer:

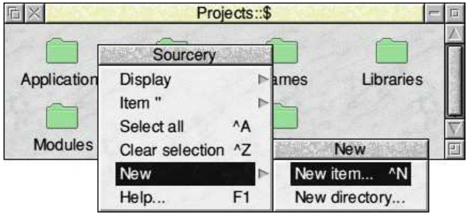
- Acorn C/C++ suite

See Appendix A for information detailing where this can be obtained from.

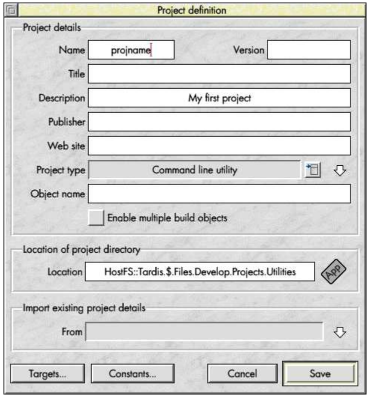
Getting started

Creating new projects in *Sourcery* is easy. Display the *Project* window by clicking **Select** on the icon bar. Also display a Filer window where the project will be physically located on disk.

Click **Menu** over the *Project* window and choose *New* followed by *New item...* from the menu:



This will display the *Project Definition* window:


Project definition

Project details

Name	projname	Version	
Title			
Description	My first project		
Publisher			
Web site			
Project type	Command line utility		
Object name			

☐ Enable multiple build objects

Location of project directory

Location: HostFS::Tardis.$.Files.Develop.Projects.Utilities

Import existing project details

From:

This window allows the initial details for the project to be specified. Not all of these details are required for each type of project. For this example just complete the following fields:

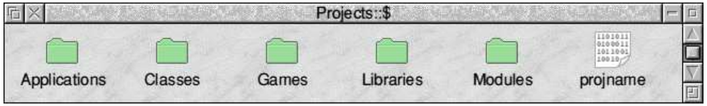
Name
The name of the project. This will be the name of the directory created to store the project files as well as the name of the target file created (unless the project is a RISC OS application). For this example enter the name *projname*.

Description
A short description of what the project does. Type *My first project*. into this field.

Project type
The list of available project types will vary according to how *Sourcery* is configured. For this simple first project choose *Command line utility.*

Location
This is the location where the project directory will be created on disk. Drag the *App* icon to the Filer window where the project is to be created.

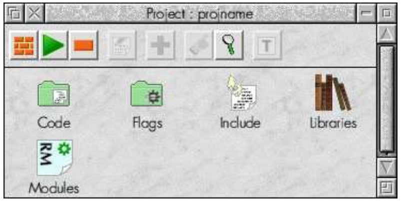
Click the *Save* button to create the project. An icon for the newly created project will appear in the *Project* window.



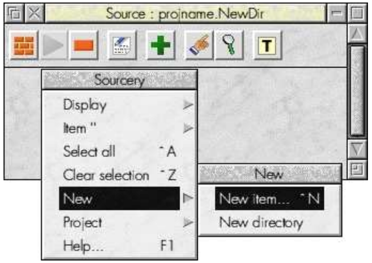
Nothing will appear in the Filer window until some source code or resources are created for the project.

Creating source code

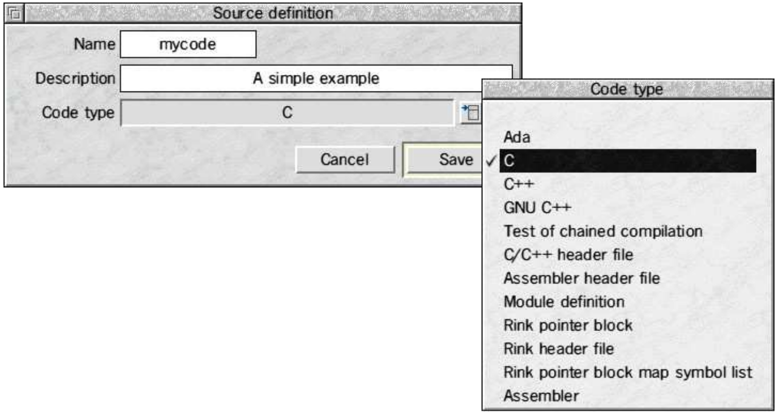
Double clicking on the newly created project in the *Projects* window will display the different components that make up the project. This will vary according to the type of project created. For this example the following will be displayed:



Double clicking on the *Code* icon will display the *Code* window which shows the source code currently defined in the project. At this point there will not be any code defined. To create some new code click **Menu** over the *Code* window and choose *New* followed by *New item...*:



The *Source definition* window will be displayed:


Source definition

Name: mycode

Description: A simple example

Code type: C

This allows the type of source code being created to be specified. *Sourcery* supports a number of different types of source code and more can be added by the user as required.

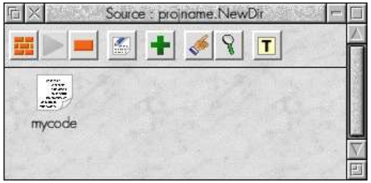
Enter the following details into this window:

Name
The name of the source code to be created. For this example type *mycode*.

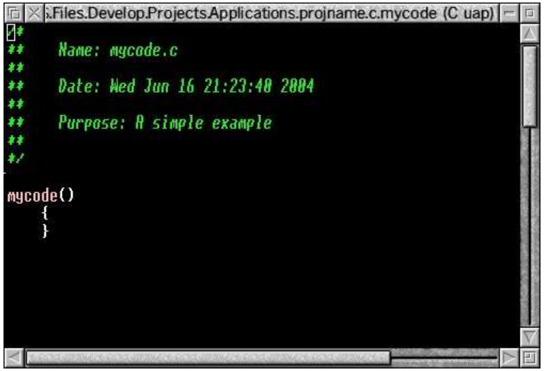
Description
A simple description of what the code does. Type *A simple example* for this first piece of code.

Code type
The available codes types will vary according to your system. For this example choose *C*.

Click the *Save* button. The new code will be created and an icon displayed in the *Source* window:



Double clicking on the code icon will load it into whatever text editor is installed on your system:


:.Files.Develop.Projects.Applications.projname.c.mycode (C uap)

```
/*
**    Name: mycode.c
**
**    Date: Wed Jun 16 21:23:40 2004
**
**    Purpose: A simple example
**
*/

mycode()
    {
    }
```

As can be seen from the example a header has been automatically generated for the code and includes the name, description and date of creation of the code.

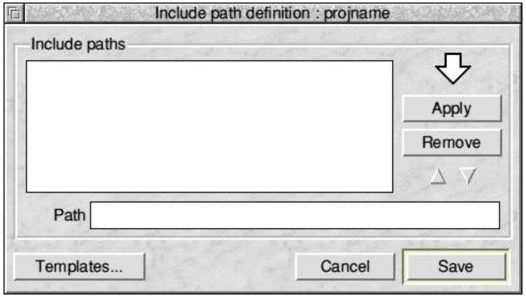
Amend everything below the header to read as follows:

```
#include <stdio.h>

int main(void)
    {
    printf("Hello world\n");
    return 0;
    }
```

Save this code for later.

Specifying include paths

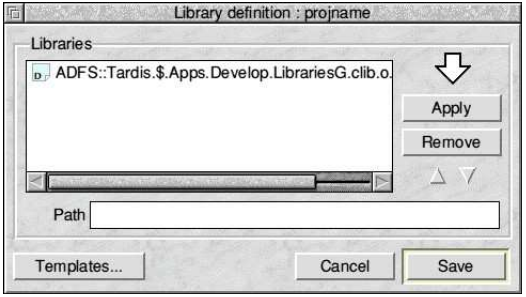
Before the project can be built it is necessary to define the include paths where the header files can be found. Double clicking on the *Include* icon in the project window will display the *Include path definition* window:



Include paths can be specified in a number of different ways. For this example drag the directory that contains the ANSI C header files for your compiler to the window and click *Save*.

Specifying libraries

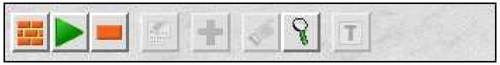
Libraries are set up in a similar way to include paths. Double clicking the *Libraries* icon will display the *Library definition* window. Drag the C library to link against to this window:


Library definition : projname
Libraries
ADFS::Tardis.$.Apps.Develop.LibrariesG.clib.o.
Apply
Remove
Path
Templates...
Cancel
Save

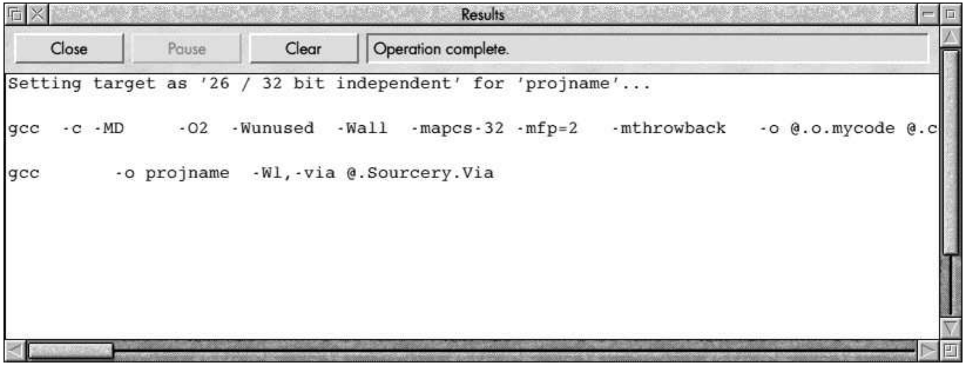
Click *Save* to save the library details. The project is now ready to be built.

Building the project

Along the top of the *Code* and *Project details* window are toolbars which allow access to certain functions:



Clicking the leftmost icon will build the project. As the build process progresses a window is displayed which shows any output generated by the compiler:



```
Setting target as '26 / 32 bit independent' for 'projname'...

gcc  -c -MD     -O2  -Wunused  -Wall  -mapcs-32 -mfp=2   -mthrowback   -o @.o.mycode @.c

gcc       -o projname  -Wl,-via @.Sourcery.Via
```

In this case the project has been successfully built. Opening the directory which contains the project source will reveal the newly created executable. Double clicking this will display *Hello world* on the screen.

Creating a GCC project

This tutorial explains how to create a project using an alternative tool chain, in this case GCC.

Assumptions

This tutorial assumes you have the following installed on your computer:

- GCC compiler and tools

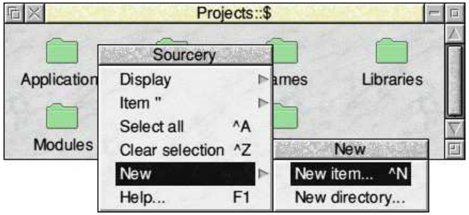
See Appendix A for information detailing where these can be obtained from.

Getting started

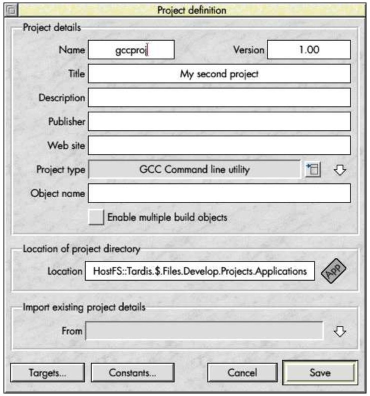
Creating projects for different tool chains is generally the same as the previous example.However, GCC is a very popular compiler so it is worth giving an example of a project using it.

Display the *Project* window by clicking **Select** on the icon bar. Also display a Filer window where the project will be physically located on disk.

Click **Menu** over the *Project* window and choose *New* followed by *New item...* from the menu:



This will display the *Project Definition* window:



Project definition

Project details

Name	gccproj	Version	1.00
Title	My second project		
Description			
Publisher			
Web site			
Project type	GCC Command line utility		
Object name			

☐ Enable multiple build objects

Location of project directory

Location	HostFS::Tardis.$.Files.Develop.Projects.Applications

Import existing project details

From	

This window allows the initial details for the project to be specified. Not all of these details are required for each type of project. For this example just complete the following fields:

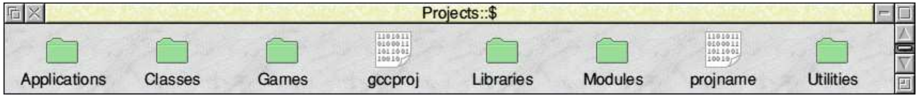
Name
The name of the project. This will be the name of the directory created to store the project files as well as the name of the target file created (unless the project is a RISC OS application). For this example enter the name *gccproj*.

Description
A short description of what the project does. Type *My second project*. into this field.

Project type
The list of available project types will vary according to how *Sourcery* is configured. For this project choose *GCC Command line utility.*

Location
This is the location where the project directory will be created on disk. Drag the *App* icon to the Filer window where the project is to be created.

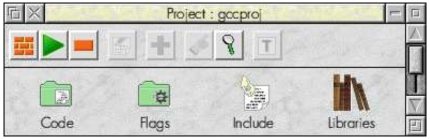
Click the *Save* button to create the project. An icon for the newly created project will appear in the *Project* window.



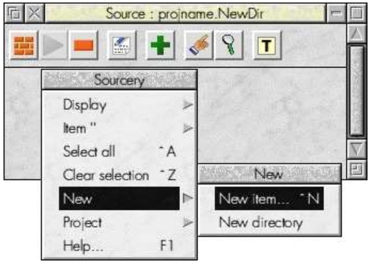
Nothing will appear in the Filer window until some source code or resources are created for the project.

Creating source code

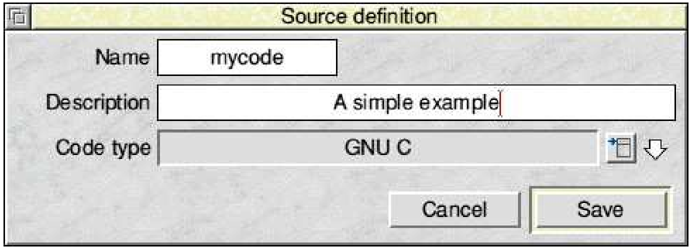
Double clicking on the newly created project in the *Projects* window will display the different components that make up the project. This will vary according to the type of project created. For this example the following will be displayed:



Double clicking on the *Code* icon will display the *Code* window which shows the source code currently defined in the project. At this point there will not be any code defined. To create some new code click **Menu** over the *Code* window and choose *New* followed by *New item...*:



The *Source definition* window will be displayed:


Source definition

Name: mycode

Description: A simple example

Code type: GNU C

This allows the type of source code being created to be specified. *Sourcery* supports a number of different types of source code and more can be added by the user as required.

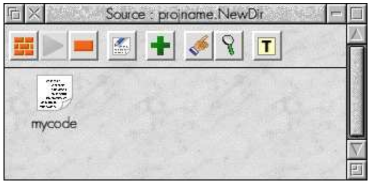
Enter the following details into this window:

Name
The name of the source code to be created. For this example type *mycode*.

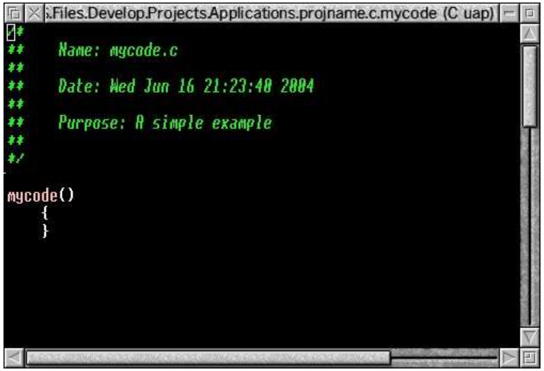
Description
A simple description of what the code does. Type *A simple example* for this piece of code.

Code type
The available codes types will vary according to your system. For this example choose *GNU C*.

Click the *Save* button. The new code will be created and an icon displayed in the *Source* window:



Double clicking on the code icon will load it into whatever text editor is installed on your system:


.Files.Develop.Projects.Applications.projname.c.mycode (C uap)

```
/*
**    Name: mycode.c
**
**    Date: Wed Jun 16 21:23:40 2004
**
**    Purpose: A simple example
**
*/

mycode()
    {
    }
```

As can be seen from the example a header has been automatically generated for the code and includes the name, description and date of creation of the code.

Amend everything below the header to read as follows:

```
#include <stdio.h>

int main(void)
    {
    printf("Hello world\n");
    return 0;
    }
```

Save this code for later.

Specifying include paths

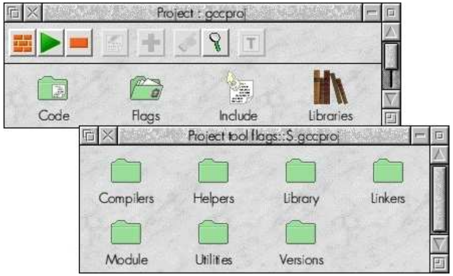
For this example GCC is actually able to find the include files it needs without us telling it where to look.

Specifying libraries

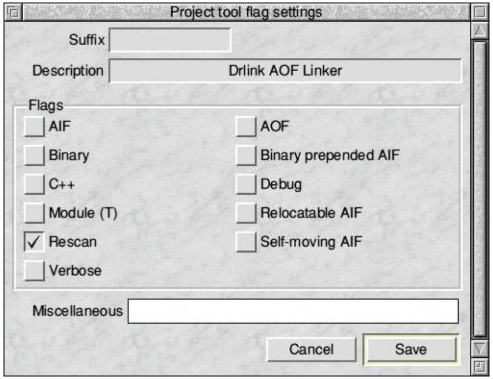
In this example GCC knows what libraries need to be supplied and will take care of that for you.

Linker flags

Depending on which version of GCC is in use, different linker flags are available. By default, *Sourcery* is configured to use version 4 of GCC in which case it should not be necessary to set any specific linker flags. This is the recommended setting. If using an earlier version however then the linker flags should be configured as follows:



Within this window, double click the *Linkers* directory, followed by *GCC* and then *drlink* to give the *Project tool flag settings* window:



Project tool flag settings

Suffix:

Description: Drlink AOF Linker

Flags

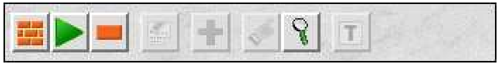
- [ ] AIF
- [ ] AOF
- [ ] Binary
- [ ] Binary prepended AIF
- [ ] C++
- [ ] Debug
- [ ] Module (T)
- [ ] Relocatable AIF
- [✓] Rescan
- [ ] Self-moving AIF
- [ ] Verbose

Miscellaneous:

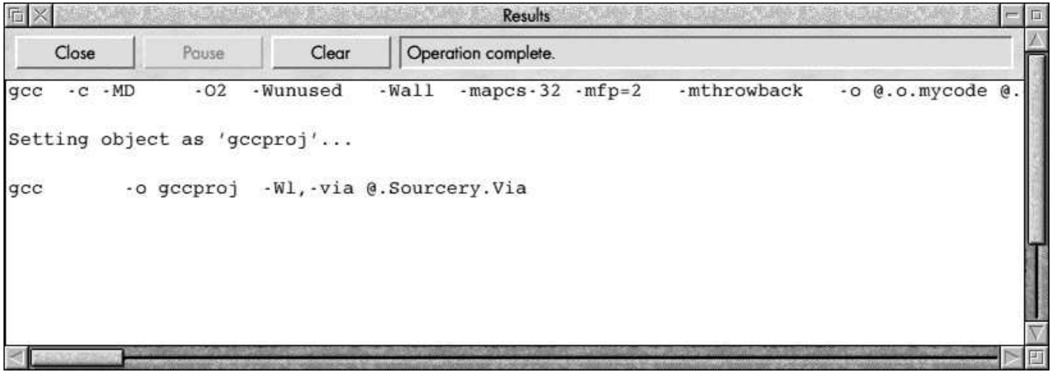
In this window make sure that the *Rescan* option is ticked and click *Save*.

Building the project

Along the top of the *Code* and *Project details* window are toolbars which allow access to certain functions:



Clicking the leftmost icon will build the project. As the build process progresses a window is displayed which shows any output generated by the compiler (this will vary depending on the version of GCC in use):



```
gcc  -c -MD     -O2  -Wunused   -Wall  -mapcs-32 -mfp=2   -mthrowback   -o @.o.mycode @.

Setting object as 'gccproj'...

gcc       -o gccproj  -Wl,-via @.Sourcery.Via
```

In this case the project has been successfully built. Opening the directory which contains the project source will reveal the newly created executable. Double clicking this will display *Hello world* on the screen.

Creating an application

The previous sections described how to create a simple command line projects. Now we will look at how to create a more complicated kind of project, a RISC OS application.

Assumptions

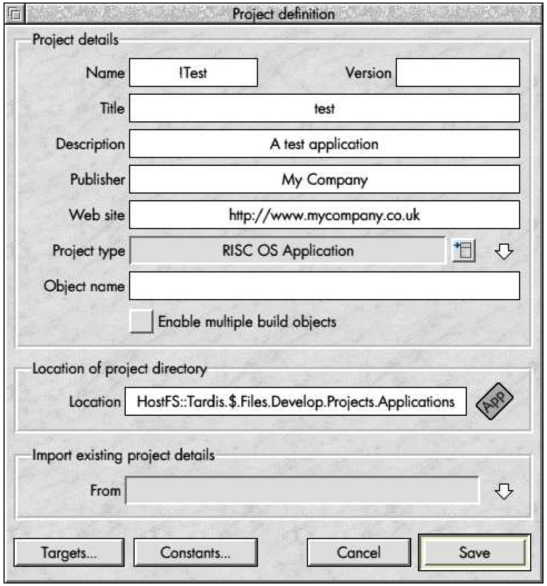
This tutorial assumes you have the following installed on your computer:

- Acorn C/C++ suite
- OSLib library

See Appendix A for information detailing where these can be obtained from.

Getting started

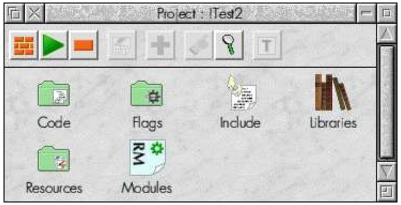
Building on what was covered before, we will create another new project with the details shown below:



Name	!Test
Version	
Title	test
Description	A test application
Publisher	My Company
Web site	http://www.mycompany.co.uk
Project type	RISC OS Application
Object name	
Enable multiple build objects	☐
Location	HostFS::Tardis.$.Files.Develop.Projects.Applications
From	

Click *Save* once the details are entered. Note that the *Location* will vary from the example shown.

Double clicking on the create project icon in *Sourcery* will display more icons in the *Project* window that the first example:



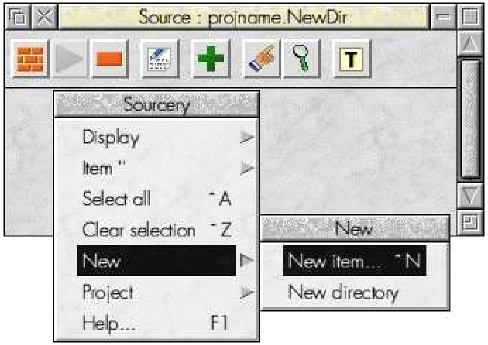
Two extra icons have been added to this window.

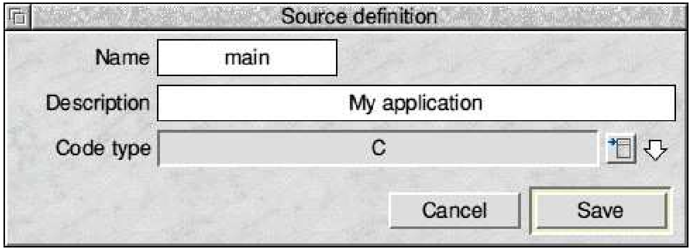
Resources
These are supporting files needed by an application to function. They include *!Run* files, *Sprites* and window templates. *Sourcery* does not limit the kind of files that can be created as resource files, nor does it limit them to being placed solely within the project directory. Sub directories can be created which contain further resource files.

Modules
Applications often rely on having specific versions of modules in order for them to work. These module versions are specified in an applications *!Run* file. Specifying and maintaining the necessary module lines can be tedious so *Sourcery* enables you to pick which modules you want and builds the *!Run* file for you.

Creating source code

Create some source code in the same way as before. Double click the *Code* icon and in the newly opened window choose *New* from the main menu:




Source definition

Name: main

Description: My application

Code type: C

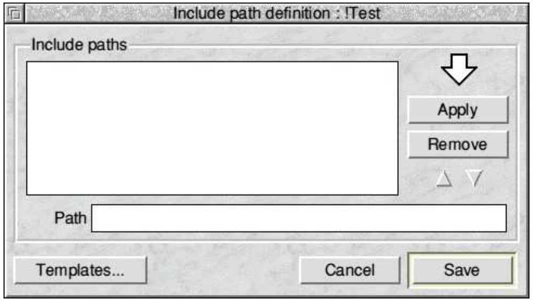
As before we will create a C source code file. Enter the code shown on the next page. Note that this is not intended to be an example of a sensible RISC OS application.

```
/*
**    Name: main.c
**
**    Date: Wed Dec  8 20:26:44 2004
**
**    Purpose: My application
**
*/

#include <string.h>
#include ″wimp.h″

int main(void)
    {
    os_error error;

    wimp_error_box_selection click;

    wimp_version_no version;

    wimp_t task;

    task = wimp_initialise(wimp_VERSION_RO30,″!Test″,NULL,&version);

    error.errnum = 0;
    strcpy(error.errmess,″Hello world″);

    click = wimp_report_error(&error,wimp_ERROR_BOX_OK_ICON,″!Test″);

    return 0;
    }
```

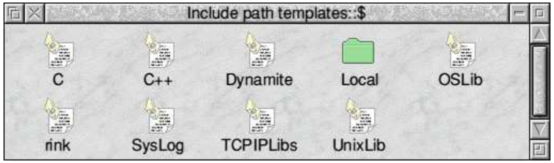
Specifying include paths

The include path that was set up in the first example involved dragging a directory from a filer window to the Include path definition window.

Display the *Include path definition* window as before:



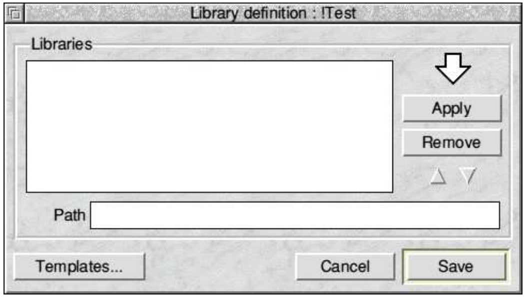
Now, from the iconbar menu choose *Templates* followed by *Include paths...*:



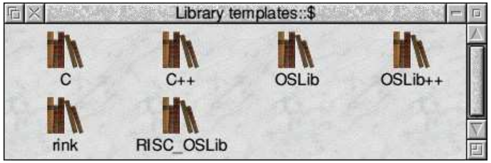
This displays a list of predefined Include paths. The *Choices* section later describes how these can be customised but for now just drag the *OSLib* icon to the *Include path definition* window. Specifying include paths in this was allows multiple physical paths to be specified at once. The definition for *OSLib* actually include OSLib and the ANSI C header paths.

Specifying libraries

In a similar way to which we have specified the include paths that we want to use, we will also specify the libraries. Display the *Library definition* window by double clicking on the *Libraries* icon:



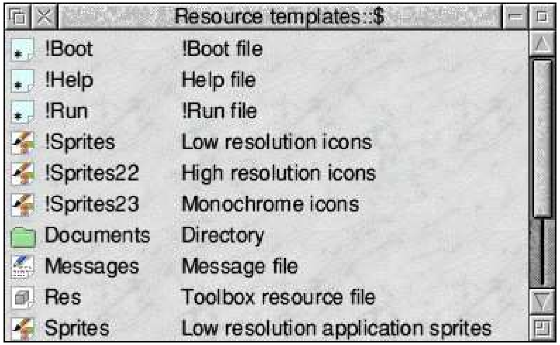
From the iconbar menu choose *Templates* followed by *Libraries...*:



Drag the *OSLib* icon to the *Library definition* window. This will include the libraries needed by OSLib as well as the Shared C Library Stubs file.

Defining resources

Double clicking the *Resources* icon allows us to define various support files needed by the application. Existing files can be directly imported into the project by dragging them from a filer window. Alternatively there are a number of template files available. Choose *Templates* followed by *Resources...* from the iconbar menu:



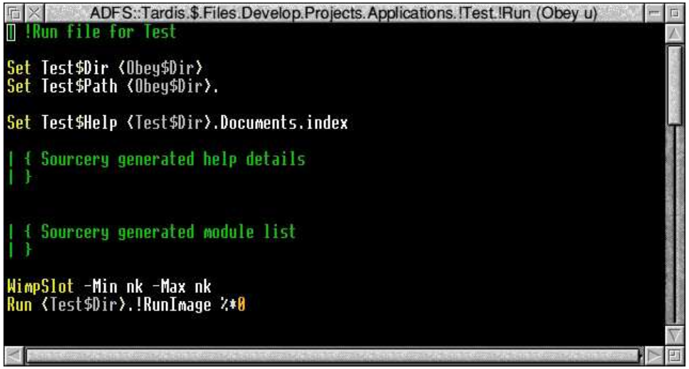
!Boot	!Boot file
!Help	Help file
!Run	!Run file
!Sprites	Low resolution icons
!Sprites22	High resolution icons
!Sprites23	Monochrome icons
Documents	Directory
Messages	Message file
Res	Toolbox resource file
Sprites	Low resolution application sprites

These files can then be dragged to the project *Resources* window.

Drag the following files from the templates window to the *Resources* window:

!Boot
!Run
!Sprites

Sourcery will try and fill in some off the details of these files when they are imported. Open the *!Run* file by double clicking on it:


ADFS::Tardis.$.Files.Develop.Projects.Applications.!Test.!Run (Obey u)

```
| !Run file for Test

Set Test$Dir <Obey$Dir>
Set Test$Path <Obey$Dir>.

Set Test$Help <Test$Dir>.Documents.index

| { Sourcery generated help details
| }


| { Sourcery generated module list
| }

WimpSlot -Min nk -Max nk
Run <Test$Dir>.!RunImage %*0
```

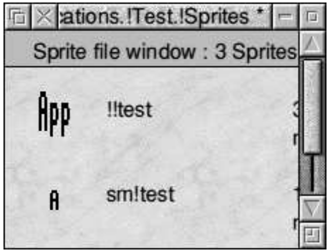
As you can see, *Sourcery* has attempted to complete certain parts of the *!Run* file automatically. The name of the application has been inserted in a number of places.

The `WimpSlot` command however needs to be amended manually to specify how much memory the application requires. In this example we can also see two special blocks that have been auto generated.

The first of these is the help details block. When the project is built this will be populated with some of the details specified in the project details window including the description, version and website for the project.

The second block is the module list block. The modules specified in the next section will be included here with the necessary `RMEnsure` and `RMLoad` commands when the project is built.

Next open the *!Sprites* file by double clicking on it.

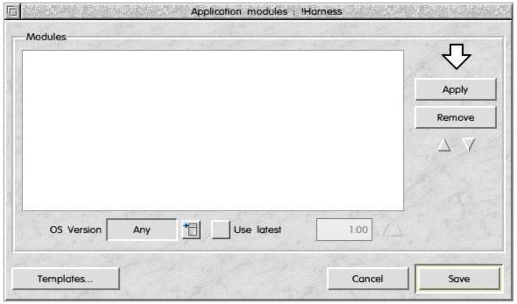


Here we can see that the sprites for the application have also been correctly named.

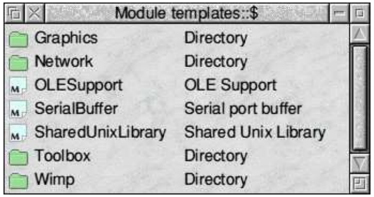
Sourcery attempts to generate resource files which are complete as possible. However, there will always be some manual editing that needs to take place on the generated file.

Modules

Double click the *Modules* icon will display a window which allows the modules that need to be loaded for the application to be run to be defined:

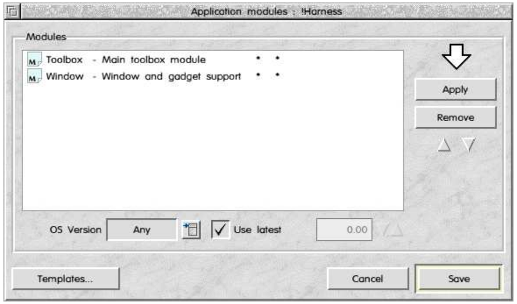


The list of available modules definitions can be seen by clicking the *Templates...* button or choosing *Templates* followed by *Modules...* from the iconbar menu:


Module templates::$

Graphics	Directory
Network	Directory
OLESupport	OLE Support
SerialBuffer	Serial port buffer
SharedUnixLibrary	Shared Unix Library
Toolbox	Directory
Wimp	Directory

Drag the required modules from the templates window to the *Application modules* window. For this example drag the *Toolbox* and *Window* modules from the *Toolbox* directory.



Toolbox - Main toolbox module • •
Window - Window and gadget support • •

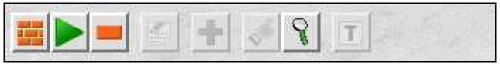
OS Version Any ✓ Use latest 0.00

Highlighting a module allows the required version number to be set. For this example leave the *Use latest* flag set to indicate that the *!Run* file will include the latest version of the module that *Sourcery* knows about.

Click *Save* to save the module list.

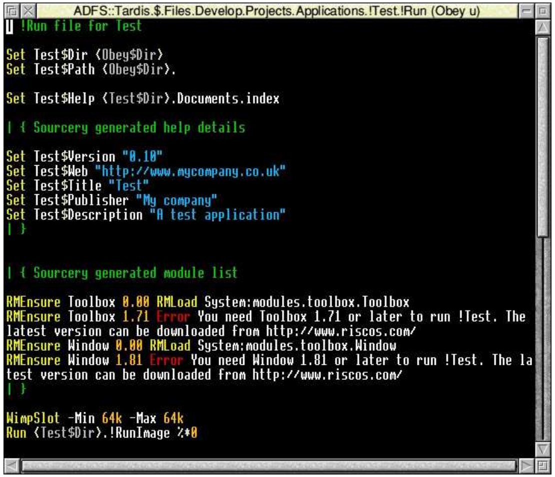
Building the project

As in the first example click the leftmost icon at the top of the *Code* or *Project* details window.



This should build the project successfully.

Before running the project open the !Run file from the Resource window. It should now look something like the following:



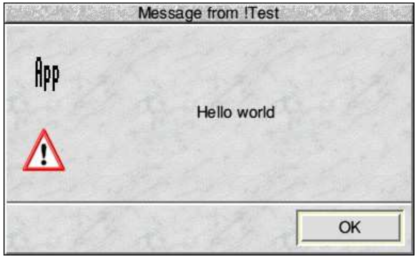
ADFS::Tardis.$.Files.Develop.Projects.Applications.!Test.!Run (Obey u)

```
| !Run file for Test

Set Test$Dir <Obey$Dir>
Set Test$Path <Obey$Dir>.

Set Test$Help <Test$Dir>.Documents.index

| { Sourcery generated help details

Set Test$Version "0.10"
Set Test$Web "http://www.mycompany.co.uk"
Set Test$Title "Test"
Set Test$Publisher "My company"
Set Test$Description "A test application"
| }


| { Sourcery generated module list

RMEnsure Toolbox 0.00 RMLoad System:modules.toolbox.Toolbox
RMEnsure Toolbox 1.71 Error You need Toolbox 1.71 or later to run !Test. The
latest version can be downloaded from http://www.riscos.com/
RMEnsure Window 0.00 RMLoad System:modules.toolbox.Window
RMEnsure Window 1.81 Error You need Window 1.81 or later to run !Test. The la
test version can be downloaded from http://www.riscos.com/
| }

WimpSlot -Min 64k -Max 64k
Run <Test$Dir>.!RunImage %*0
```

As you can see, the help and module blocks have been filled in with the information specified in the *Project definition* and *Application modules* windows.

Running the project

Locate the project directory in the Filer window where it was created and double click on it to run it. The following window should appear on the screen:


Message from !Test

Hello world

Importing an existing project

Sourcery allows existing projects to be imported and tries to do as much of the work as possible. In this example we will look at importing one of the example applications that comes with DeskLib, and compiling and building it using GCC.

Assumptions

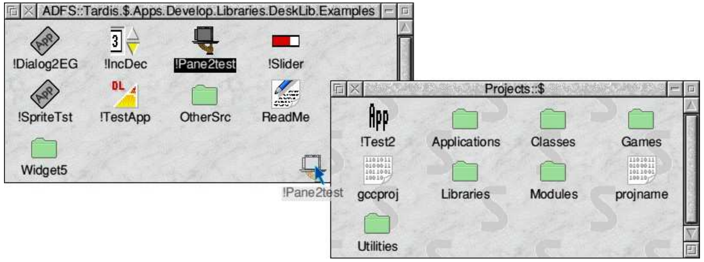
This tutorial assumes you have the following installed on your computer:

- GCC compiler and tools
- DeskLib library

See Appendix A for information detailing where these can be obtained from.

Getting started

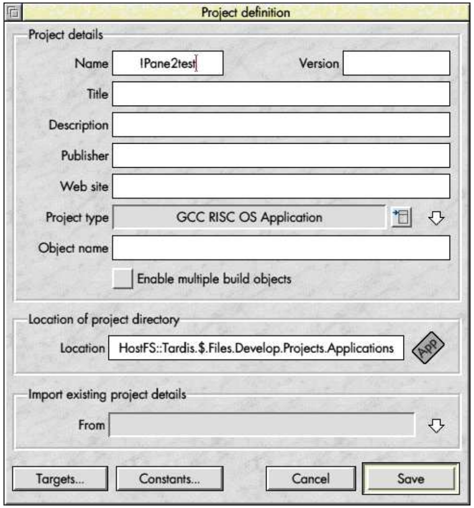
Make sure the *Projects* window is open and drag the project to be imported to it. For this example we will use *!Pane2test*:



The *Project definition* window will open. The From field will contain the pathname of the *!Pane2test* application. Complete the following fields:

Name:	!Pane2test
Project type:	GCC RISC OS Application

Drag the *Location* icon to the filer window where you want the imported *!Pane2test* application to reside. This should not be the same location as where it currently resides.


Project definition

Project details

Name: !Pane2test
Version:
Title:
Description:
Publisher:
Web site:
Project type: GCC RISC OS Application
Object name:
Enable multiple build objects

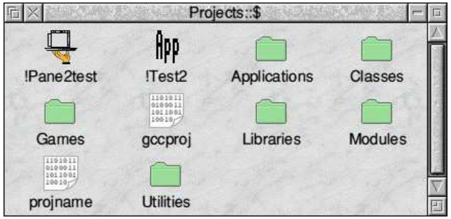
Location of project directory

Location: HostFS::Tardis.$.Files.Develop.Projects.Applications

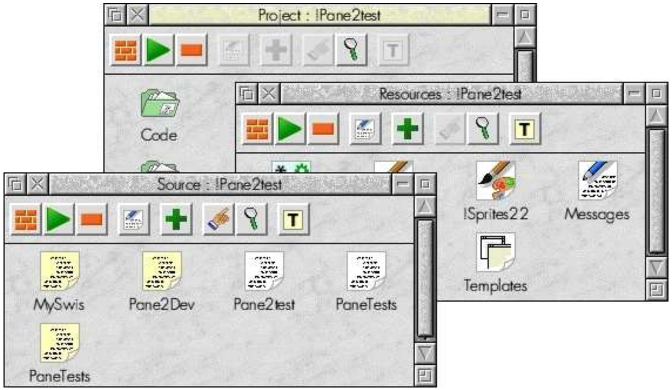
Import existing project details

From:

Click the *Save* button. After a few seconds a new icon will appear in the *Projects* window.



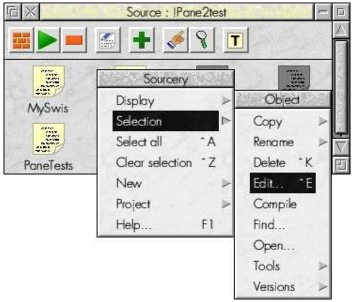
Double clicking on the *!Pane2test* icon will open the *Project* window. Double clicking on the *Code* and *Resources* icons will display the source code and resources that have been imported into *Sourcery*:



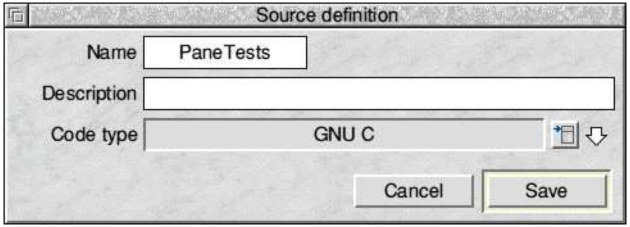
Sourcery will do the best it can to identify and import resources for an application. However, opening the original *!Pane2test* application will show that the *Templates3D* file failed to be recognised and imported. Simply drag this file to the *Resources* window and click *Save* in the *Resource definition* window that appears.

Setting the code type

In this example we can see that three header files and two source files have been imported. Because in this example we are going to use GCC to compile the code we need to make sure the correct code type is being used. Highlight the Pane2test and PaneTests source code and choose Edit from the Selection sub menu:



The source definition window will appear for each item of source code:



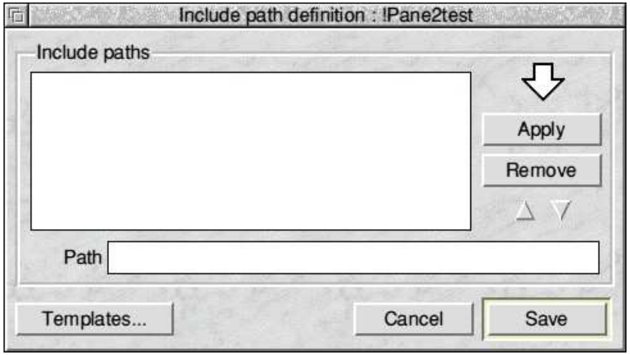
Source definition

Name	PaneTests
Description	
Code type	GNU C

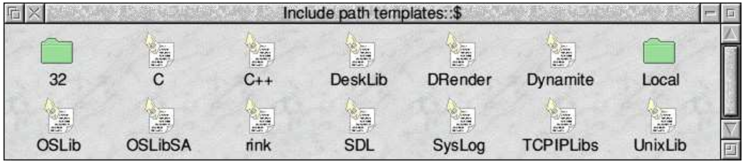
Ensure that the *Code type* is set to *GNU C* for both and click the *Save* button.

Specifying include paths

Display the *Include path definition* window by double clicking on the *Include* icon in the *Project* window:



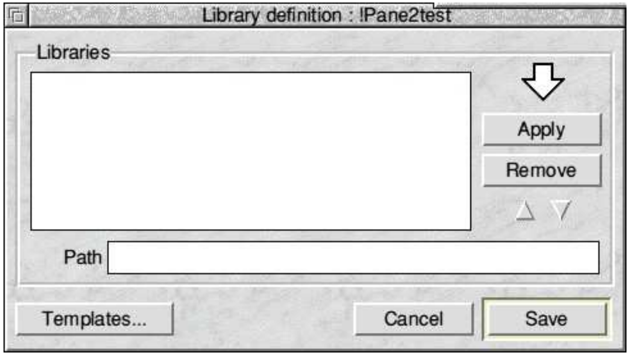
Now, from the iconbar menu choose *Templates* followed by *Include paths...*:



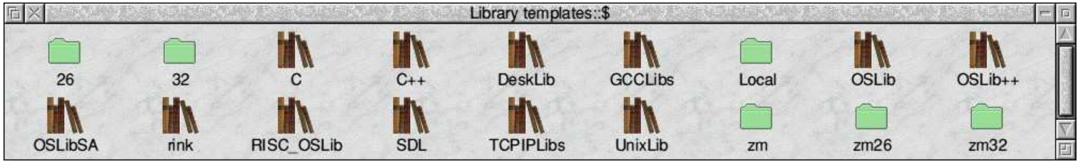
This displays a list of predefined Include paths. Drag the *DeskLib* icon to the *Include path definition* window. Click the *Save* button.

Specifying libraries

Display the Library definition window by double clicking on the Libraries icon in the Project window:



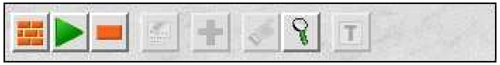
From the iconbar menu choose *Templates* followed by *Libraries...*:



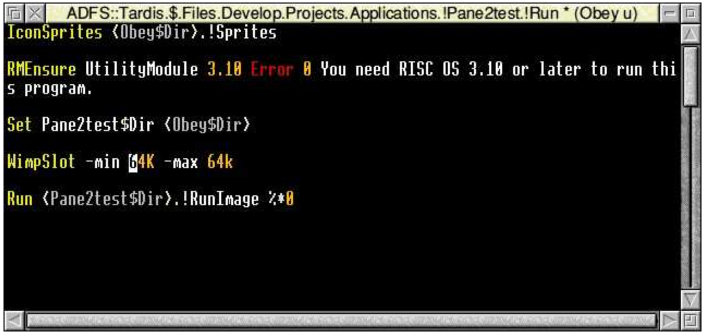
Drag the *DeskLib* icon to the *Library definition* window. Click the *Save* button.

Building the project

Click the leftmost icon at the top of the *Code* or *Project* details window to build the project:



Before running the project open the *!Run* file from the *Resource* window. It should now look something like the following:


ADFS::Tardis.$.Files.Develop.Projects.Applications.!Pane2test.!Run * (Obey u)

```
IconSprites <Obey$Dir>.!Sprites

RMEnsure UtilityModule 3.10 Error 0 You need RISC OS 3.10 or later to run thi
s program.

Set Pane2test$Dir <Obey$Dir>

WimpSlot -min 64K -max 64k

Run <Pane2test$Dir>.!RunImage %*0
```

For this example we used GCC which has resulted in a larger executable size. The line that reads:

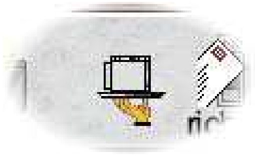
```
    WimpSlot -min 64K -max 64K
```

should be changed to:

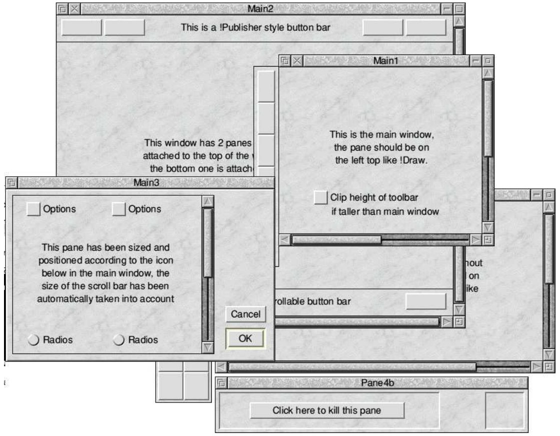
```
    WimpSlot -min 264K -max 264K
```

Running the project

Locate the project directory in the Filer window where it was created and double click on it to run it. The !Pane2test icon will appear on the iconbar:



The application will then behave in the expected way:



This window has 2 panes
attached to the top of the
the bottom one is attach

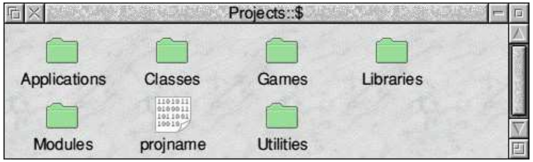
This is the main window,
the pane should be on
the left top like !Draw.

This pane has been sized and
positioned according to the icon
below in the main window, the
size of the scroll bar has been
automatically taken into account

Reference

Projects

The *Projects* window is displayed by clicking **Select** on the icon bar icon:



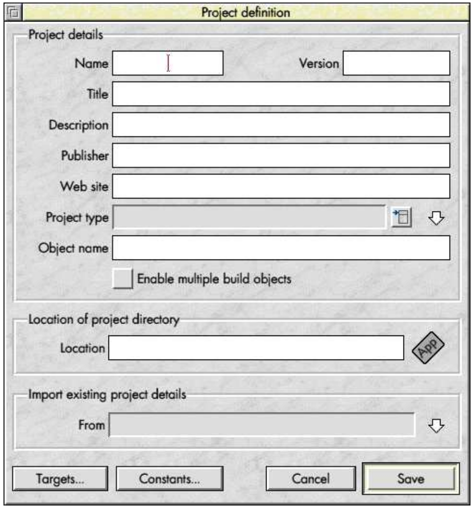
This window allows new projects to be created and existing projects to be edited. Double clicking on a project will display some of the components that make up the project.

Exporting a project

A project can be exported from *Sourcer*y either in a state that is suitable for distribution or in source code form. When exported in a state suitable for distribution the compiled binary and any defined resources will be exported but the source code and files that *Sourcery* creates to store project details will not. To export a project it should be dragged to a Filer window from the *Sourcery Projects* window. A prompt will appear asking whether the project should be exported complete with source or just runtime files. Clicking OK will export the whole project, clicking Cancel will only export the runtime files.

Project definition

To create a new project choose *New* from the *Projects* menu to display the *Project definition* window:



Name

The name of the project. This is the name that is displayed in the *Projects* window. It is also the name of the executable or library file that will be created when the project is built.

Version

The version number that will appear in a program info dialog if one if one is present and will also be included in help details in the !Run file.

Title
The text that is used when help information is displayed as the name or title of the project.

Description
A short description that describes the purpose of the project. This is displayed in the *Projects* window when the Full info view is selected.

Publisher
The author or company producing the software. This is used by the help system if available.

Web site
The address of the project or company home page on the Internet where information about the project can be found.

Project type
Sourcery supports a number of different project types. These have predefined settings to handle the building of RISC OS applications, command line utilities, modules and libraries as well as others that can be defined by the user.

Object name
By default, the name of any object file or executable created when a project is built will be the name of the project or, for applications, *!RunImage*. Specifying a name here will cause this name to be used instead. The specified name can contain a number of symbols that will be replaced at build time:

$(name)
The name as specified in the *Name* field.

$(version)
The version as specified in the *Version* field.

$(target)
The target of the object currently being built. For an explanation of *Targets*, see later.

This is particularly of use when building shared libraries with GCC which have a specific naming convention. For example, the library ODBC might have an object name as follows:

```
lib$(name).so.$(version)
```

which might give an output file name something like:

```
libODBC.so.1.2.3
```

or

```
libODBC/so/1/2/3
```

under RISC OS. The translation of the . to / is handled automatically where necessary.

Enable multiple build objects
If this is switched on it is possible to generate more than one binary object when a project is built. For example, when creating a library it may be beneficial to have a test harness that is used to test the functionality of the library. In this situation both the library and test harness could be built from within the one project. If this is switched on an extra icon appears in the *Project* window. This is covered in more detail in a later section.

Location

The location of the project where it is to be stored on disc. The location can be typed but it is easier to drag the icon to the directory where the project is to be saved.

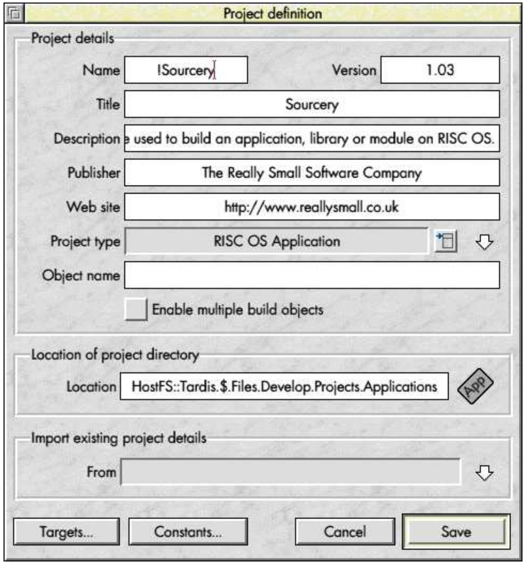
Import
Sourcery can import source and resources from projects that are built using make (although the makefile itself will not be imported) and projects that have been defined in RPM. If a project is being imported it should be dragged to this field.

Once these fields have been completed clicking the *Save* button will create a new project.

The *Constants* button allows constants to be defined for the project. This is covered later.

Editing a project definition

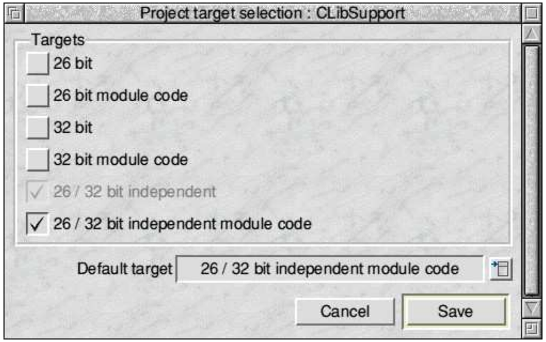
A project definition can be edited by clicking **Menu** over the project in the *Projects* window and choosing *Item* followed by *Edit...*. The *Project definition* window is then displayed with the existing project details.



Project definition
Project details
Name
!Sourcery
Version
1.03
Title
Sourcery
Description
e used to build an application, library or module on RISC OS.
Publisher
The Really Small Software Company
Web site
http://www.reallysmall.co.uk
Project type
RISC OS Application
Object name
Enable multiple build objects
Location of project directory
Location
HostFS::Tardis.$.Files.Develop.Projects.Applications
App
Import existing project details
From
Targets...
Constants...
Cancel
Save

Targets

Sourcery enables more than one target file to be created from the same set of source code. This is especially useful when building a library that will be linked with normal applications and with relocatable modules which require certain compiler options to be used when code is compiled. The *Targets* window allows the user to specify which different targets will be built for a project:


Project target selection : CLibSupport

Targets
- [ ] 26 bit
- [ ] 26 bit module code
- [ ] 32 bit
- [ ] 32 bit module code
- [✓] 26 / 32 bit independent
- [✓] 26 / 32 bit independent module code

Default target: 26 / 32 bit independent module code

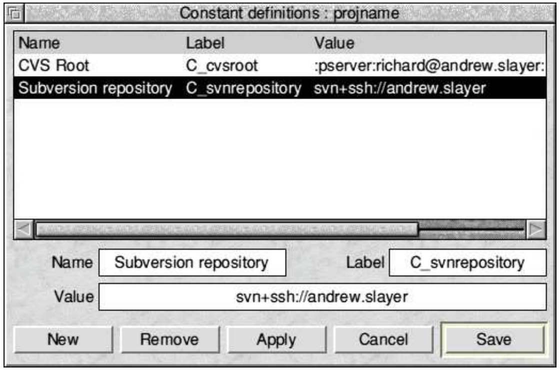
The available targets can be customised as required, see the *Choices* section later in this guide. Any greyed out targets cannot be deselected, they will always be available to be built. This can also be configured when changing the list of available targets.

Default target
A default target must be specified for the project. This is the target that will normally be generated when the project is built. The remaining targets are generated by explicitly selecting them to be built. This is covered later.

Clicking *Save* will store the selected targets for the project.

Constants

Projects can have constants associated with them that can be used when defining the commands that are used to build and maintain source code and resources (see *Tools* later). The *Constants* window allows the name, label and value of the constant to be defined:


Constant definitions : projname

Name	Label	Value
CVS Root	C_cvsroot	:pserver:richard@andrew.slayer:
Subversion repository	C_svnrepository	svn+ssh://andrew.slayer

Name: Subversion repository
Label: C_svnrepository
Value: svn+ssh://andrew.slayer

The example shows project wide constants that have been defined for CVS and Subversion source control. Normally only one of these would be defined. A naming protocol exists for *Constants* to minimise confusion when they are used in *Tool* commands. The *Constant* label should always be prefixed with C_.

A *Constant* can be amended by highlighting it and changing the fields at the bottom of the window:

Name
The name of the *Constant*. This is only used in this window.

Label
The label that will be used in the *Tool* command to specify the use of the *Constant*.

Value
The value that will replace the label in the *Tool* command when the command is executed.

The following buttons are available:

New
Adds a new *Constant* to the list.

Remove
Removes the currently highlighted *Constants* from the list.

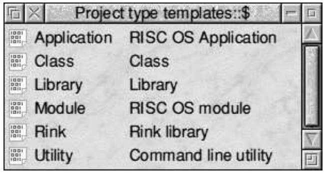
Apply
Applies changes made to name, label and value to the currently highlighted *Constant*.

Cancel
Closes the window without saving any changes.

Save
Closes the window and saves any changes made to the *Constants*.

Project type templates

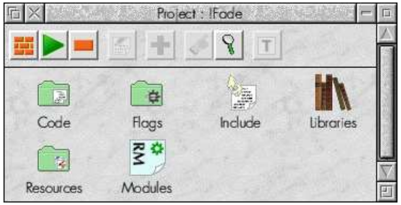
Clicking **Menu** over the icon bar icon and choosing *Templates* followed by *Project types...* will display a list of available project types. The required project type can be dragged from this window to the *Project definition* window as an alternative to using the *Project type* field.



Application	RISC OS Application
Class	Class
Library	Library
Module	RISC OS module
Rink	Rink library
Utility	Command line utility

Project details

Double clicking on a project in the *Projects* window will display the *Project details* window:



This window displays the different components that make up a project. Different project types can have different combinations of component types so not all projects have the components shown here. Double clicking on any of these components will open the appropriate window.

Code
Every project will have code. This is the source code that will be compiled to form the executable or library.

Include
A lot of programming languages have the concept of include files. These are files that are included into the source that is written to provide common definitions or constants. The project has to be told where to find the include files that it will use.

Flags
The tools that are used to compile the source code often have options that can control the way the code is compiled. These flags can be set on a per project basis.

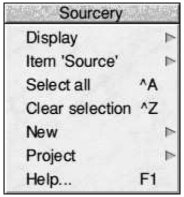
Libraries
Libraries are pieces of compiled code, often supplied by third parties, which can be linked into a project to provide useful functionality.

Resources
Resources are normally only used when creating a RISC OS application. They are the !Boot, !Run, sprite and any other files that are required for the application to run.

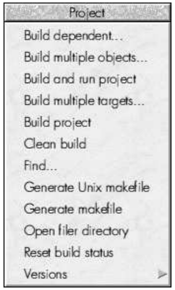
Modules
Modules are also only available when creating a RISC OS application. They allow the user to specify the modules that appear in the applications !Run file without having to manually edit the file.

Note that most of the standard operations such as copying, deleting and creating new items have no effect in this window.

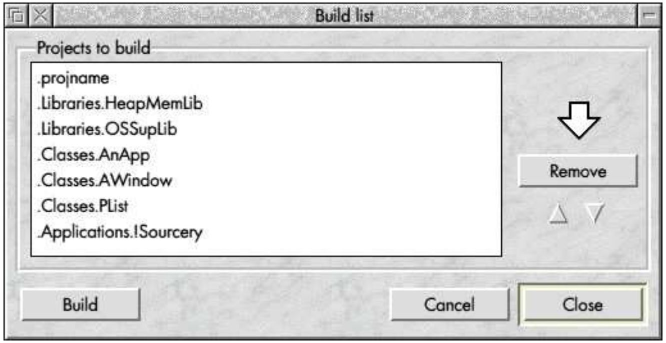
The standard menu also an additional entry.



Project
This sub menu will display a list of operations that can be carried out on the whole project. The contents of the sub menu can be customised by the user and will therefore vary from what is shown here:



Build dependent...
When this item is chosen a window will appear that allows any projects that the current project is dependent on to be automatically built one after another:



.projname
.Libraries.HeapMemLib
.Libraries.OSSupLib
.Classes.AnApp
.Classes.AWindow
.Classes.PList
.Applications.!Sourcery

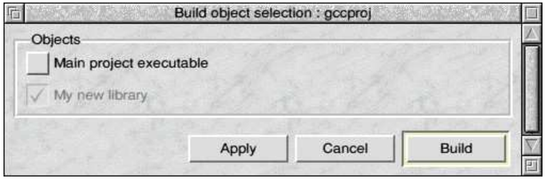
When the window is opened for the first time all of the dependent projects for the current project are automatically added to the list. All of the libraries included by each project are be examined and if they are defined within *Sourcery* they will be added to the list. This operation works recursively to try and satisfy all dependencies.

Further projects can be added by dragging them from the *Projects* window to the arrow in this dialog. Projects can be reordered in the list by using the up and down arrows and removed altogether by using the *Remove* button.

Clicking the *Build* button will build each project in turn, stopping if there is an error.

This feature is particularly useful if a number of libraries are included in a project and a header file is changed. Using this feature will rebuild any code that is affected by the change.

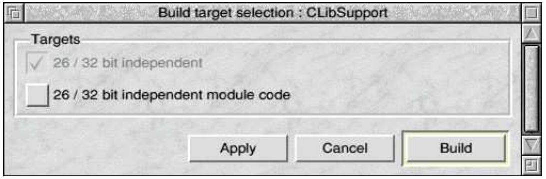
Build multiple objects...
This item will display a window showing all of the available build objects that are available for the project:



Selecting the objects to be built and clicking *Build* will build each object in turn. Clicking *Apply* will store the settings but will not actually build the objects. If an object is greyed out it is defined as always being built.

Build and run project
The same as *Build* except that for Application type projects *Sourcery* will launch the project after it has built.

Build multiple targets...
This item will display a window showing all of the available build targets that are available for the project:

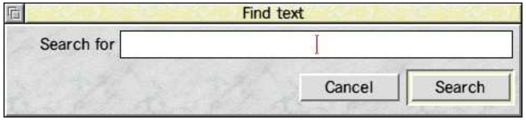

Build target selection : CLibSupport
Targets
26 / 32 bit independent
26 / 32 bit independent module code
Apply
Cancel
Build

Selecting the targets to be built and clicking *Build* will build each target in turn. Clicking *Apply* will store the settings but will not actually build the targets. Note that any objects marked to be built in the *Build object selection* window will be built when multiple targets are built in this way.

Build project
Choosing this will build the project. Any code that needs compiling will be compiled and the resultant code combined into the final binary file for the project.

Clean build
This will remove all object files from a project when chosen and force all code to be recompiled the next time the project is built.

Find
Clicking *Find* will open a window that allows the source code to be searched for an expression.



Enter the text to be searched for and click *Search*. Any results will be displayed in a standard throwback window.

Generate makefile
Attempts to generate a makefile so that make or amu can be used to build the project.

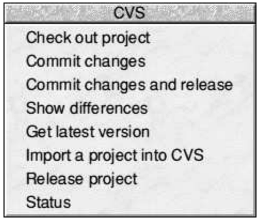
Generate Unix makefile
Attempts to generate a makefile suitable for using on Unix type operating systems.

Open filer directory
Opens the filer window that contains the project directory.

Reset build status
Used to reset the status of a project so that Sourcery no longer thinks the project is being built. Note this was added to work around an issue that should be fixed as of version 1.10 or later.

Versions
Gives access to CVS or Subversion version control tools. Note that these commands work on the whole project when accessed via this menu.

CVS
The following are available from the CVS menu:



Check out project
Checks out the project so changes can be made to the source code and resources.

Commit changes
Commits any changes back to the repository. A prompt will appear requesting the message that should be associated with the changes.

Commit changes and release
Commits any changes back to the repository and releases the project so other users can make changes.

Show differences
Shows any differences between the local working copy of the project and that stored in the repository.

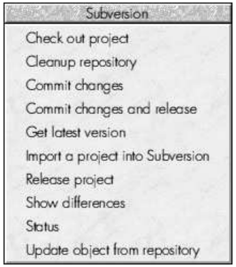
Get latest version
Gets the latest version of the project files.

Import a project into CVS
Imports a complete project into CVS.

Release project
Release the project so that other may work on it without committing any changes.

Status
Shows the status of the project.

Subversion
The following are available from the CVS menu:



Check out project
Checks out the project so changes can be made to the source code and resources.

Cleanup repository
Cleans up the working copy.

Commit changes
Commits any changes back to the repository. A prompt will appear requesting the message that should be associated with the changes.

Commit changes and release
Commits any changes back to the repository and releases the project so other users can make changes.

Show differences
Shows any differences between the local working copy of the project and that stored in the repository.

Get latest version
Gets the latest version of the project files.

Import a project into CVS
Imports a complete project into CVS.

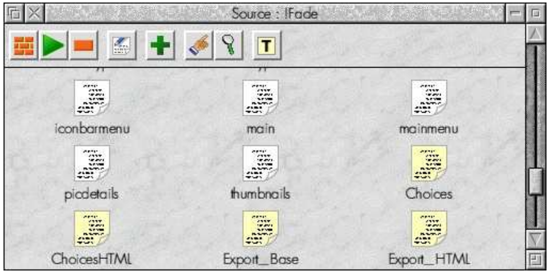
Release project
Release the project so that other may work on it without committing any changes.

Status
Shows the status of the project.

Update object from repository
Updates working copy with changes from repository.

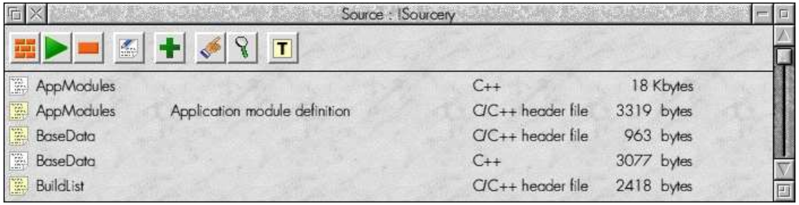
Code

Double clicking on the *Code* icon in the *Project details* window will display the source code that is defined for the project:



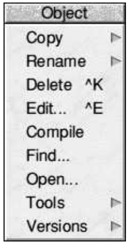
Double clicking on a piece of code will load it into a text editor. Code that *Sourcery* decides is a header file is displayed in a different colour to other code. Also, code where the underlying file is read only will be displayed faded. This is to accommodate version control systems that might change the access attributes of a file.

When using the Full Info display in this window some additional information is shown. As well as the code name and description, the type of the code and the size of the underlying file is shown.



AppModules		C++	18 Kbytes
AppModules	Application module definition	C/C++ header file	3319 bytes
BaseData		C/C++ header file	963 bytes
BaseData		C++	3077 bytes
BuildList		C/C++ header file	2418 bytes

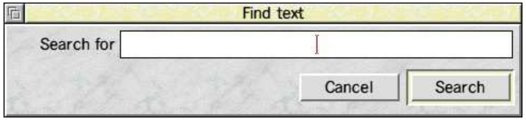
The *Item* menu has some extra options. The contents of this menu can be customised by the user and will therefore vary from what is shown here:



The extra menu items are as follows:

Compile
Compiles any highlighted code in the project without linking it.

Find
Clicking Find will open a window that allows the source code to be searched for an expression.



Enter the text to be searched for and click Search. Any results will be displayed in a standard throwback window.

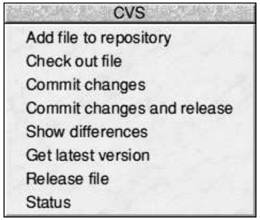
Open
This is used when you want to edit multiple pieces of code. Rather than double clicking on each in turn, they are all selected at once and choosing this item will then open them all for editing.

Tools
The Tools sub menu gives access to the following tool:

Touch
Sometimes it is necessary to trigger the recompilation of code without having made any changes to it. This is often referred to as 'touching' the code. Select the code to be touched and choose this.

Versions
Gives access to CVS or Subversion version control tools. Note that these commands work on the highlighted items of code when accessed via this menu.

CVS
The following are available from the CVS menu:



Check out file
Checks out the code so changes can be made.

Commit changes
Commits any changes back to the repository. A prompt will appear requesting the message that should be associated with the changes.

Commit changes and release
Commits any changes back to the repository and releases the project so other users can make changes.

Show differences
Shows any differences between the local working copy of the code and that stored in the repository.

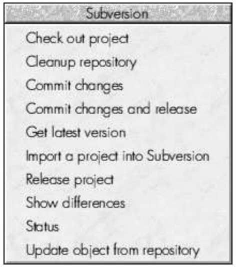
Get latest version
Gets the latest version of the source code.

Release file
Release the code without committing any changes so that others may work on it .

Status
Shows the status of the code.

Subversion

The following are available from the CVS menu:



Check out project
Checks out the project so changes can be made to the source code and resources.

Cleanup repository
Cleans up the working copy.

Commit changes
Commits any changes back to the repository. A prompt will appear requesting the message that should be associated with the changes.

Commit changes and release
Commits any changes back to the repository and releases the project so other users can make changes.

Show differences
Shows any differences between the local working copy of the project and that stored in the repository.

Get latest version
Gets the latest version of the project files.

Import a project into CVS
Imports a complete project into CVS.

Release project
Release the project so that other may work on it without committing any changes.

Status
Shows the status of the project.

Update object from repository
Updates working copy with changes from repository.

Adding new code

New code can be added in two different ways.

New code

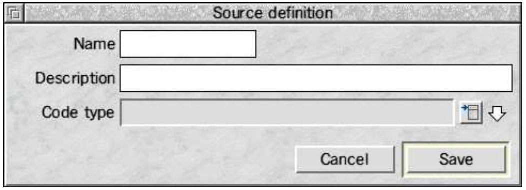
Choosing *New* from the menu or clicking the *New* icon will display the *Source definition* window covered in the next section.

Filer

Dragging a text file either with a suitable suffix or included in a suitable suffix type directory to the *Sources* window will open a *Source definition* window and allow the file to be imported

Source definition

To create new source code choose *New* from the *Code* menu to display the Source *definition* window:



Name

The name of the source code. This is the name that is displayed in the *Code* window. It does not have to be the internal name of the piece of code. In fact, there is nothing to stop there being multiple functions or procedures within a single source code file.

Description

A short description that describes the purpose of the code. This is displayed in the *Code* window when the Full info view is selected.

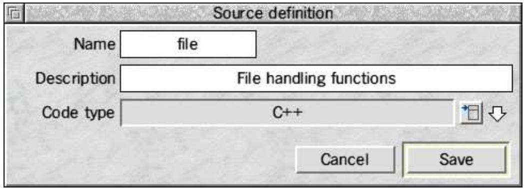
Code type

Sourcery supports a number of different code types. These have predefined templates that are populated when the code is created. New code types can be added by the user.

Once these fields have been completed clicking the *Save* button will create a new code.

Editing a source definition

A source definition can be edited by clicking **Menu** over the code in the *Code* window and choosing *Item* followed by *Edit...*. The *Source definition* window is then displayed with the existing code details.

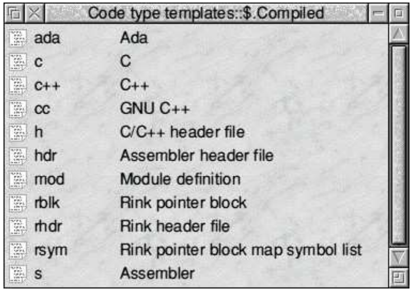


Source definition

Name	file
Description	File handling functions
Code type	C++

Code type templates

Clicking **Menu** over the icon bar icon and choosing *Templates* followed by *Code types...* will display a list of available code types. The required code type can be dragged from this window to the *Source definition* window as an alternative to using the *Code type* field. Clicking the *Templates...* icon on the toolbar will also display the code types.



Code type templates::$.Compiled

ada	Ada
c	C
c++	C++
cc	GNU C++
h	C/C++ header file
hdr	Assembler header file
mod	Module definition
rblk	Rink pointer block
rhdr	Rink header file
rsym	Rink pointer block map symbol list
s	Assembler

!RunImage source definition

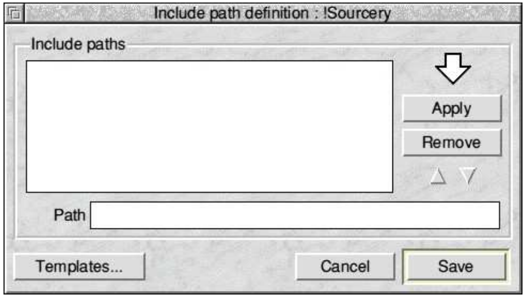
If a piece of source is created with the name *!RunImage*, and it is defined as being an interpreted code type such as BASIC, then it is treated differently by *Sourcery*.

Source code is normally located within a separate directory within the project directory. For compiled languages this source code is translated into a single file called *!RunImage* which is the main application binary. Interpreted languages work a little differently in that the code is not translated until it is run**.** Because this translation stage does not take place, no *!RunImage* file is produced which would mean that applications written in interpreted languages in *Sourcery* would have no *!RunImage* file.

To get around this limitation, *Sourcery* treats any interpreted file called *!RunImage* differently, and automatically copies it to the project directory when the project is built or the source code compiled.

Include path definition

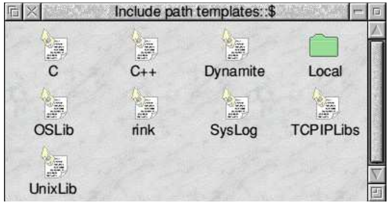
Double clicking the *Include* icon in the *Project details* window will display this window:



There are three ways Include paths can be specified:

Include path templates

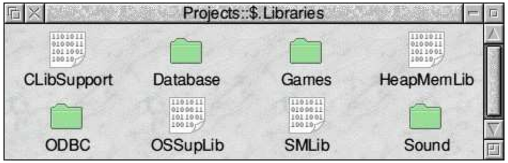
Clicking the *Templates...* button or clicking **Menu** over the icon bar icon and choosing *Templates* followed by *Include paths...* will display a list of available include paths. These can be dragged to the definition window. Note that these templates can be set up to contain multiple paths.



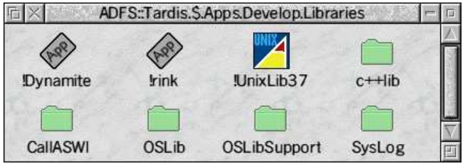
For example, the *OSLib* template in the window shown has the OSLib includes paths and the standard ANSI C include paths defined.

Include paths defined in this way have a small version of the icon used in the *Include paths template* window next to them.

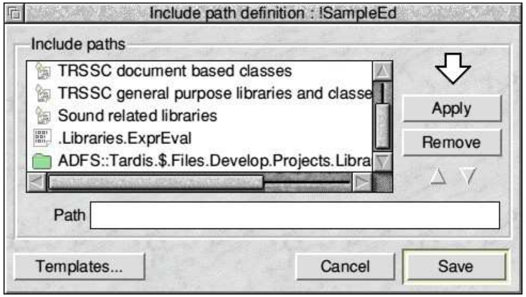
Projects
The second way in which an include path can be specified is to drag a project from the *Projects* window. This will allow any header files defined in this project to be included. A small project icon will appear next to an include path added in this way.



Filer
The final way to add an include path is to drag a directory from a Filer window. Any header files in this directory can then be used. A small directory icon will appear next to the included directory.



A definition with a selection of include paths sources is shown below:



TRSSC document based classes
TRSSC general purpose libraries and classe
Sound related libraries
.Libraries.ExprEval
ADFS::Tardis.$.Files.Develop.Projects.Libra

Editing an include path

Clicking on an entry will display it in the *Path* field at the bottom of the window. Non project include paths can be edited in this field. Click *Apply* to store the changes.

Deleting an include path

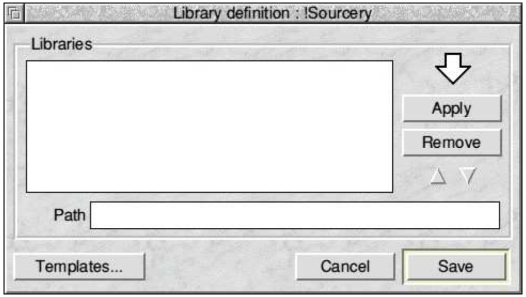
Highlight the include paths to be deleted and click the *Remove* button. The include paths will disappear from the list. Note that this will not be permanent until *Save* is clicked.

Changing the order of include paths

The position of the include path in the list can be changed through the use of the up and down arrows. This will affect the order in which the paths are passed to the compilation command when any code is compiled.

Library definition

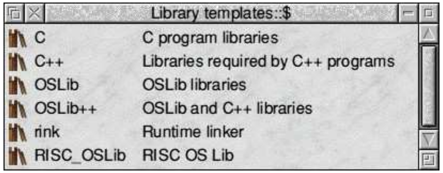
Double clicking the *Libraries* icon in the *Project details* window will display this window:



There are three ways Libraries can be specified:

Library templates

Clicking the *Templates...* button or clicking **Menu** over the icon bar icon and choosing *Templates* followed by *Libraries...* will display a list of available libraries. These can be dragged to the definition window. Note that these templates can be set up to contain multiple libraries.

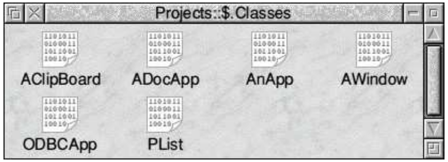


Library templates::$

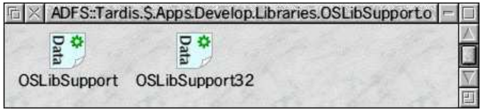
C	C program libraries
C++	Libraries required by C++ programs
OSLib	OSLib libraries
OSLib++	OSLib and C++ libraries
rink	Runtime linker
RISC_OSLib	RISC OS Lib

For example, the *OSLib++* template in the window shown has the OSLib, C++ and C libraries defined.
Libraries defined in this way have a small version of the icon used in the *Libraries* template window next to them.

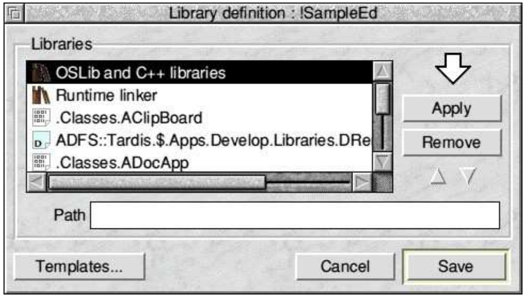
Projects
The second way in which a library can be specified is to drag a project from the *Projects* window. This will allow any header files defined in this project to be included. A small project icon will appear next to an include path added in this way.



Filer
The final way to add an include path is to drag a library from a Filer window.



A definition with a selection of include paths sources is shown below:



Editing a library

Clicking on an entry will display it in the *Path* field at the bottom of the window. Non project libraries can be edited in this field. Click *Apply* to store the changes.

Deleting a library

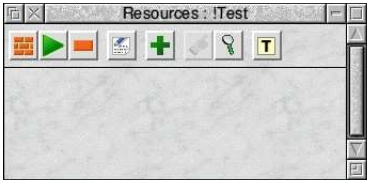
Highlight the libraries to be deleted and click the *Remove* button. The libraries will disappear from the list. Note that this will not be permanent until *Save* is clicked.

Changing the order of libraries

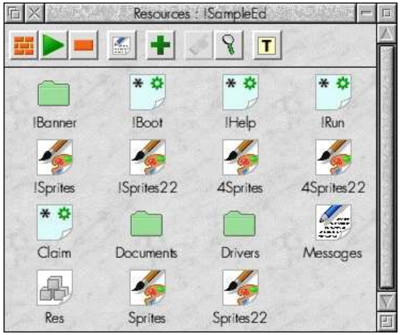
The position of the library in the list can be changed through the use of the up and down arrows. This will affect the order in which the libraries are passed to the linking stage of the build process.

Resources

Double clicking on the *Resources* icon in the *Project details* window will display the resources for the project. This is normally only available when creating a RISC OS application.

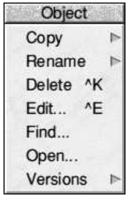


This window will initially be blank when first displayed.



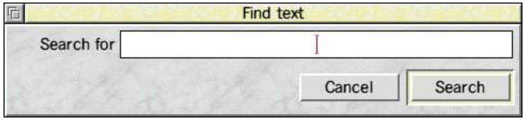
Once some resources have been defined the contents will look much like the contents of an application directory.

The *Item* sub menu has some extra options when opened over this window. The contents of this menu can be customised by the user and will therefore vary from what is shown here:



The extra menu items are as follows:

Find
Clicking Find will open a window that allows the resources to be searched for an expression.



Enter the text to be searched for and click *Search*. Any results will be displayed in a standard throwback window.

Open
This is used when you want to edit multiple resource files. Rather than double clicking on each in turn, they are all selected at once and choosing this item will then open them all for editing.

Versions
Gives access to CVS or Subversion version control tools. Note that these commands work on the highlighted resource items when accessed via this menu.

CVS
The following are available from the CVS menu:

Check out file
Checks out the file so changes can be made.

Commit changes
Commits any changes back to the repository. A prompt will appear requesting the message that should be associated with the changes.

Commit changes and release
Commits any changes back to the repository and releases the project so other users can make changes.

Show differences
Shows any differences between the local working copy of the file and that stored in the repository.

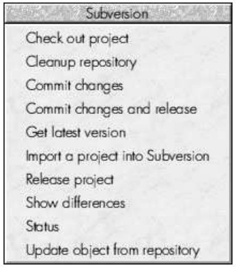
Get latest version
Gets the latest version of the resource file.

Release file
Release the file without committing any changes so that others may work on it.

Status
Shows the status of the file.

Subversion

The following are available from the CVS menu:



Check out project
Checks out the project so changes can be made to the source code and resources.

Cleanup repository
Cleans up the working copy.

Commit changes
Commits any changes back to the repository. A prompt will appear requesting the message that should be associated with the changes.

Commit changes and release
Commits any changes back to the repository and releases the project so other users can make changes.

Show differences
Shows any differences between the local working copy of the project and that stored in the repository.

Get latest version
Gets the latest version of the project files.

Import a project into CVS
Imports a complete project into CVS.

Release project
Release the project so that other may work on it without committing any changes.

Status
Shows the status of the project.

Update object from repository
Updates working copy with changes from repository.

Adding new resources

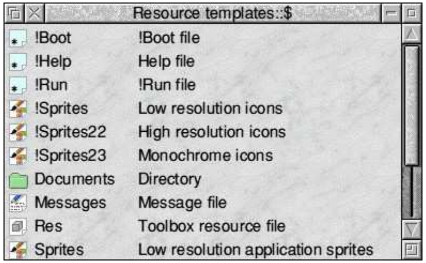
A new resource can be added in three different ways.

New resource

Choosing New from the menu or clicking the New icon will display the *Resource definition* window covered in the next section.

Resource templates

Clicking the Templates icon on the toolbar or clicking **Menu** over the icon bar icon and choosing *Templates* followed by *Resources...* will display a list of available resources.



Resource templates::$

!Boot	!Boot file
!Help	Help file
!Run	!Run file
!Sprites	Low resolution icons
!Sprites22	High resolution icons
!Sprites23	Monochrome icons
Documents	Directory
Messages	Message file
Res	Toolbox resource file
Sprites	Low resolution application sprites

The required resource can be dragged from this window to the *Resources* definition window.

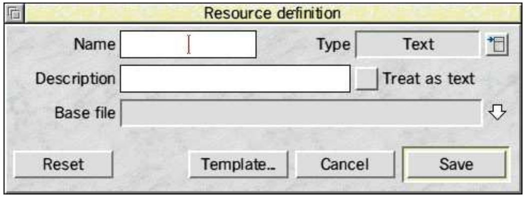
Filer
Finally, any file can be dragged from a Filer window to be added as a resource. This will open the *Resource definition* window to allow additional details to be entered.

If a directory is dragged to the *Resources* window then a message will appear asking if the directory should be imported en masse, in which case a single *Resource definition* window will be opened, or individually, in which case a *Resource definition* window will be opened for each file contained in the directory or any sub directories. The imported directory hierarchy will be maintained.

Editing a resource
Double clicking on a resource in this window will load it into the relevant editor.

Resource definition

To create a new resource choose *New* from the *Resources* menu to display the *Resource definition* window:



Name
The name of the resource. This is the name that is displayed in the *Resources* window and the name of the file on disc.

Description
A short description of the purpose of the resource. This is displayed in the *Resources* window when the Full info view is selected.

Base file
Drag an external file here to import it as the resource. Dragging a file here when the *Resource definition* is being edited and saving will overwrite the existing resource.

Type
The file type of the resource can be set to any valid RISC OS file type by selecting one from this list.

Treat as text
If this is selected then when the resource is edited it will be loaded into a text editor rather than the editor defined by its file type. This is useful for Obey, HTML and other files which normally do something other than launching an editor when they are double clicked.

Once these fields have been completed clicking the *Save* button will create a new code.

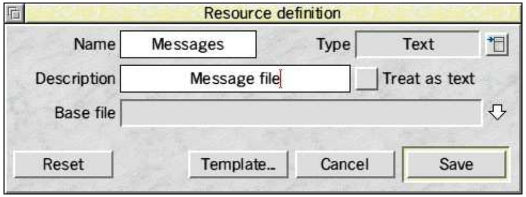
There are two further buttons on this window:

Reset
If a file has been dragged to this window for import and appears in the *Base file* field then clicking *Reset* will remove the reference to it.

Template...
Clicking this will open the resource in the relevant editor for editing. This button will only be active if the resource has first been saved.

Editing a resource definition

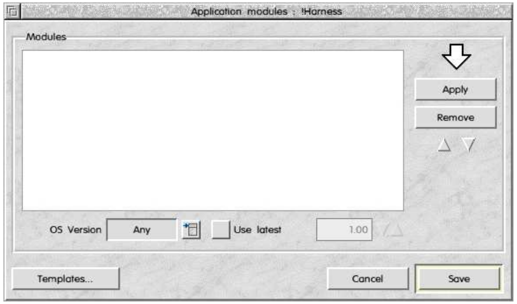
A resource definition can be edited by clicking **Menu** over the resource in the *Resources* window and choosing *Item* followed by *Edit...* The *Resource definition* window is then displayed with the existing resource details.


Resource definition

Name: Messages
Type: Text
Description: Message file
Treat as text
Base file:

Modules

Double clicking the *Modules* icon in the *Project details* window will display this window:

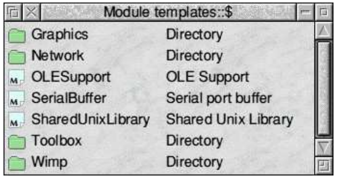


This window allows a list of external modules that the project needs to be maintained. This list is then used to build the !Run file for the application and adds the relevant checks to ensure the correct version of each module is loaded. Next to the description of the module are displayed the minimum version of the module required (* for the latest available) and the version of RISC OS that this is relevent to (* indicates any version).

If necessary it is possible to specify a different list of modules for different versions of RISC OS. This facility is provided mainly to circumvent the problems caused by the different versions of RISC OS numbering Toolbox modules inconsisently.

Adding a module

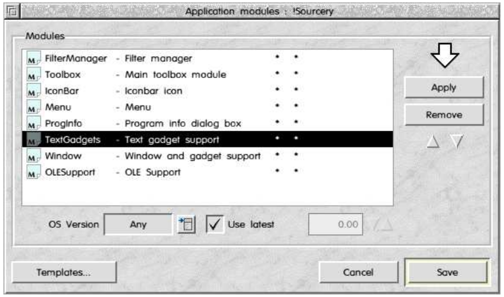
A module is added to this window by dragging an entry from the modules template window. To display the template window click the *Templates...* button or choose *Templates* followed by *Modules...* from the icon bar menu.


Module templates::$

Graphics	Directory
Network	Directory
OLESupport	OLE Support
SerialBuffer	Serial port buffer
SharedUnixLibrary	Shared Unix Library
Toolbox	Directory
Wimp	Directory

Module version

Applications often require that a specific version of a module is present. Highlighting a module allows the version number to be specified.


Application modules : !Sourcery

Modules

FilterManager	- Filter manager	•	•
Toolbox	- Main toolbox module	•	•
IconBar	- Iconbar icon	•	•
Menu	- Menu	•	•
ProgInfo	- Program info dialog box	•	•
TextGadgets	- Text gadget support	•	•
Window	- Window and gadget support	•	•
OLESupport	- OLE Support	•	•

OS Version: Any

✓ Use latest

0.00

Normally it is best to select the *Use latest* option. This will use the latest version number that *Sourcery* knows about (this can be configured using Choices - see later). Sometimes however it may be necessary to require a different version number. Unselecting *Use latest* will activate the *Version* field. The required version number can then be entered. Click *Apply* to store the change and *Save* to make the change permanent.

Specifying a module for a specific version of RISC OS

Located next to the *Use latest* option is the *OS Version* list. If required a different version of a module can be specified for a specific version of RISC OS. Normally this should be set to *Any*, but if an application uses the Toolbox it may be necessary to specify a different version for RISC OS 5 compared to other versions. *Any* versions will always be used unless there is an overriding RISC OS version specific entry.

Only the major version of RISC OS is considered when performing this check.

If RISC OS version specific modules are specified then the *!Run* file is built slightly differently. A series of additional files are created within the *!RunFiles* directory with one being created for each specified version of RISC OS plus a general version. The main *!Run* file then determines which of these to invoke.

Deleting a module

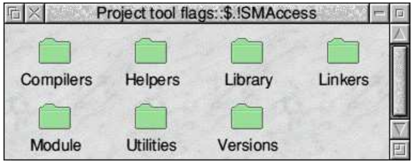
Highlight them modules to be deleted and click the *Remove* button. The modules will disappear from the list. Note that this will not be permanent until *Save* is clicked.

Changing the order of modules

The position of the module in the list can be changed through the use of the up and down arrows.

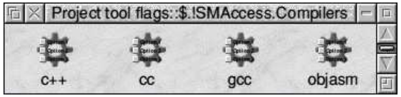
Project flags

Double clicking the *Flags* icon in the *Project details* window will display the *Flags* window:



This window displays all of the tools that *Sourcery* uses to compile, build or any number of other things. These tools often have options that can be turned on and off and which require setting on a per project basis. Through this window the flags for all these tools can be set although a particular project may not use all the tools. The settings only apply to that project and can be different for every project.

The tools are split into groups. Within the *Compilers* directory will be the list of compiler tools:

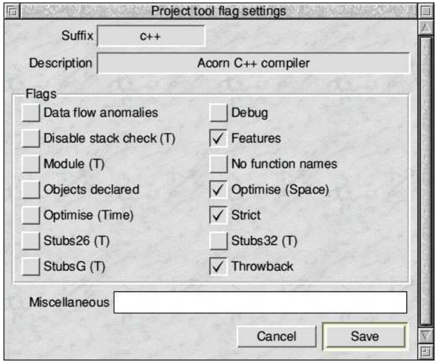


Note that these groups and the tools within them will vary depending on how *Sourcery* is configured.

Note that some tools may be greyed out. This is because it is possible to force *Sourcery* to use system wide settings for a tool.

Double clicking on one of these tools will display the flag settings window.

Project tool flag settings


Project tool flag settings

Suffix	c++
Description	Acorn C++ compiler

Flags

Data flow anomalies	Debug
Disable stack check (T)	✓ Features
Module (T)	No function names
Objects declared	✓ Optimise (Space)
Optimise (Time)	✓ Strict
Stubs26 (T)	Stubs32 (T)
StubsG (T)	✓ Throwback

Miscellaneous

Suffix
This is a display only field that shows the suffix that source files must have to be processed by this tool.

Description
A short description of which tool is being used for this suffix.

Flags
This section varies depending on which tool is being configured. Knowledge of the tool is required to set these options sensibly.

The default settings that are supplied with *Sourcery* should be suitable to begin with. These can then be changed as the user becomes more experienced.

Flags that have (T) following their name should not be selected unless the user is certain of what they are doing. These flags are used to generate different build targets for a project and get set according to which target is currently being built. They are included in this window for the advanced user.

Miscellaneous
This is some text that will be embedded into the command generated when the tool is run. It allow for options to be specified for a project that would not make sense as flags.

For example, defining certain constants might fall into this category.

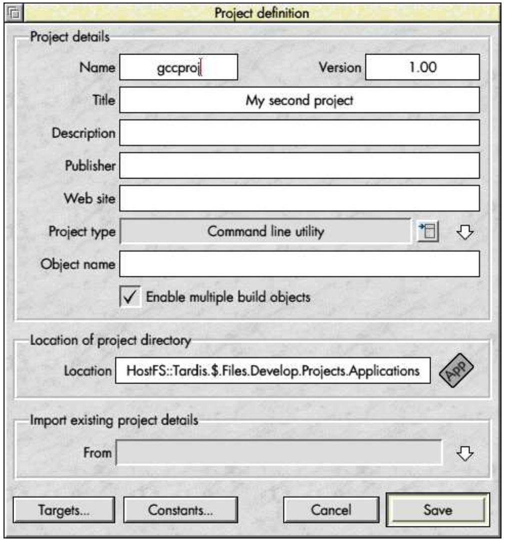
Clicking *Save* will save the tool flag settings.

Multiple build objects

As mentioned briefly earlier in this manual, it is possible to specify multiple build objects for a project which allows more than one binary object to be built from one project.

Note that this feature is intended for more advanced users.

To activate this feature display the *Project definitio*n window for a project:


Project definition

Project details

Name: gccproj
Version: 1.00
Title: My second project
Description:
Publisher:
Web site:
Project type: Command line utility
Object name:
[✓] Enable multiple build objects

Location of project directory

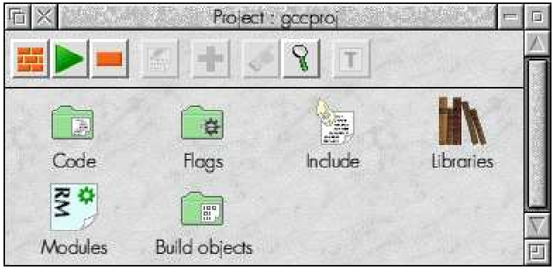
Location: HostFS::Tardis.$.Files.Develop.Projects.Applications

Import existing project details

From:

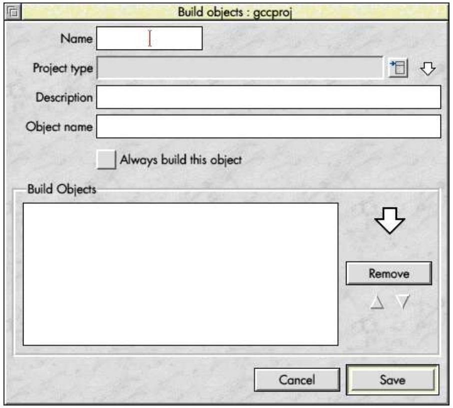
Within this window switch on the *Enable multiple build objects* option and click *Save*.

When this option is activated a new icon is displayed in the *Project* window as shown:



Double clicking the *Build Objects* icon displays the *Build Objects* window:

This window will be empty to begin with. Choosing *New* from the main menu will allow a *Build Object* to be defined by displaying the *Build Object* window:



This window contains the following:

Name
The name of the build object. This is the name that will be given to the binary file created and replaces the *Project name* for standard projects. The behaviour for RISC OS applications is the same, in which case this name will be ignored and a name of *!RunImage* used.

Project type
As it is possible to create multiple build objects, *Sourcery* needs to know how to create each different object. The list of available types here is the same as in the *Project definition* window and enables *Sourcery* to invoke different tools, for example a linker or a library generation tool, to create the object.

Description
A brief description of the build object. This only appears in the *Build Objects* window.

Object name
An optional field that specifies the name of the object file or executable that is created. See *Object name* under *Project definition* for full details.

Always build this object
If this is switched on then the object will always be built when the project is built.

Build Objects
This area of the window allows the source files that are going to be used to build the object to be specified. To do this simply drag the required files from the *Code* window to this window. This tells *Sourcery* to take the object code produced from these files when they are compiled and include it in this *Build Object*. One source file can be included in many build objects within a project.

If a file is no longer required to be part of a build object then it can be highlighted and the *Remove* button clicked. This will only remove it from the build object and will not delete the source from the project.

The order of the source files can also be changed by highlighting a file and using the up and down arrows to alter the position.

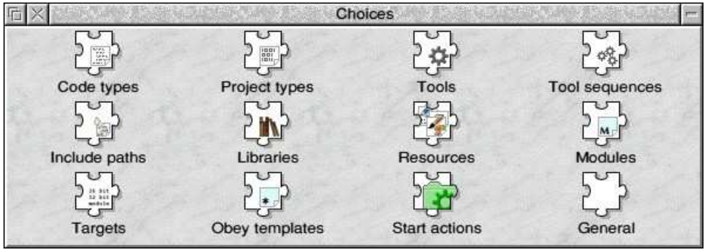
Master project type
Be aware that when multiple build objects are activated it is necessary to define at least one build object. The default, or master, project type will not generate a binary object.

It is also worth considering what the master project type should be for the project. The master project type will be used to determine which icons are available in the *Project* window. Therefore, if a library and a command line utility are to be built from a project the master project type should be set to *Command line* as this will make the *Include* and *Libraries* icons available.

Building multiple objects
See the section covering the *Project* menu in the *Project definition* window for details on how to build multiple objects.

Choices

Choosing *Choices...* from the icon bar menu will display the *Choices* window:



This window allows nearly all areas of *Sourcery* to be configured according to the tools and libraries that the developer has installed on their system.

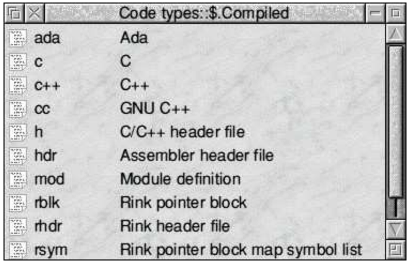
All of these icons except *General* give a Filer like window when clicked on.

The *Item* sub menu for most *Choices* windows has one extra item:

Reset to default
Selecting this item will reset the highlighted objects back to their default settings so they are the same as when *Sourcery* was first installed.

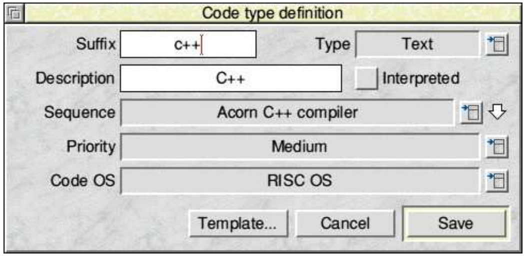
Code types

This window allows the types of code that Sourcery understands to be configured.


Code types::$.Compiled

ada	Ada
c	C
c++	C++
cc	GNU C++
h	C/C++ header file
hdr	Assembler header file
mod	Module definition
rblk	Rink pointer block
rhdr	Rink header file
rsym	Rink pointer block map symbol list

Choosing *New* from the menu will display the *Code type definition* window:


Code type definition

Suffix: c++
Type: Text
Description: C++
Interpreted: ☐
Sequence: Acorn C++ compiler
Priority: Medium
Code OS: RISC OS

This window allows the details for a code type to be defined.

Suffix
The suffix of the file associated with the code type. For example, C files normally have a suffix of c, C++ files may have a suffix of c++ or cc.

Filetype
The filetype of the underlying file. This will normally be Text for compiled languages but some interpreted languages will have other types.

Interpreted
If selected this specifies that the language is an interpreted one. This means that when the project is exported for distribution any interpreted source code will also be exported.

Description
A short description of the code type.

Sequence
Tool sequences are described later in more details. This is basically the list of commands that will be run when the code is compiled. If no sequences is specified the code type is assumed to be some sort of header file.

Priority
Occassionally it can be useful to have code compile in a specific order. This might be necessary if a piece of code is dependent on the output produced from another file. The priority controls the order in which code is compiled. It should normally be set to *medium*.

Code OS
It is possible for *Sourcery* to run commands remotely on other computers running other operating systems. If one is to make use of this feature it is important to remember that the way in which source files are named can be slightly different.
For example, on RISC OS source code is named as follows:

```
      <pathname>.<source type suffix>.<source file>
```

Whilst on Unix it would be as follows:

```
      <pathname>.<source file>/<source type suffix>
```

Note that the RISC OS directory separator has been used in both these examples.

The key point is that on RISC OS source files reside in a directory which specifies the type of source. On Unix the type is concatenated to the name as a suffix.

In order for *Sourcery* to successfully run commands remotely the correct form of naming convention must be used. Choosing the operating system on which the compilation commands will be run here will ensure that the files are named correctly.

Note that it is also necessary for the code type to use a *Tool Sequence* that has been set up to run a *Tool* specifically defined for the remote OS and that the source files must reside on a network drive that both operating systems can access.

Click *Save* to save the definition.

Template...
This button allows a template to be defined that will be populated when code of this type is created. The template for a C++ file is shown below:

```
/*
**    Name: $name.$suffix
**
**    Date: $date
**
**    Purpose: $description
**
*/

$name()
    {
    }
```

There are some special tokens that can be embedded in the template. These will be replaced when code is created.

$application
The name of the project that has created the code.

$date
The date when the code was created.

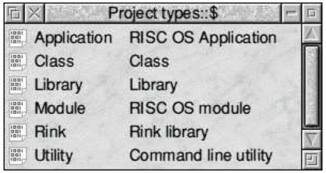
$description
The description entered for the code.

$name
The name of the code.

$suffix
The suffix of the code.

Project types

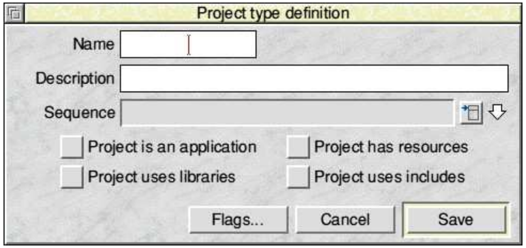
This allows the different types of project that *Sourcery* can create to be edited:



Project types::$

Application	RISC OS Application
Class	Class
Library	Library
Module	RISC OS module
Rink	Rink library
Utility	Command line utility

Choosing *New* from the *Project type* menu will display the *Project type definition* window:



This window allows the project type details to be defined:

Name
The name of the project type.

Description
A short description of the project type.

Sequence
Tool sequences are described later in more details. This is basically the list of commands that will be run when the project is built.

Project is an application
If this is selected then the project will be built with the executable name *!RunImage*.

Project uses resources
If this is selected then the project will include *Resources* in the *Project details* window.

Project uses libraries
If this is selected then the project will include *Libraries* in the *Project details* window.

Project uses includes
If this is selected then the project will include *Libraries* in the *Project details* window.

Clicking *Save* will save the project type definition.

Flags
The different tools that are used to build a project can have numerous options which can be switched on or off. Clicking this button will display the *Project flags* window which allows default settings for this type of project to be set up. The working of this window are described elsewhere.

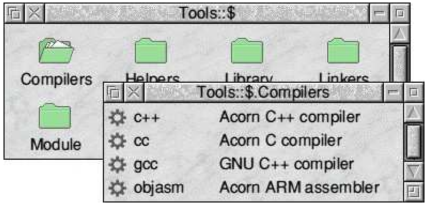
See also the section on *Targets* for a description of how the different flag settings interact with each other.

Resources

Projects that are defined as applications can have resource files attached to them. This window allows the default set of resource files to be maintained. It works in the same way as the *Resources window* in a project.

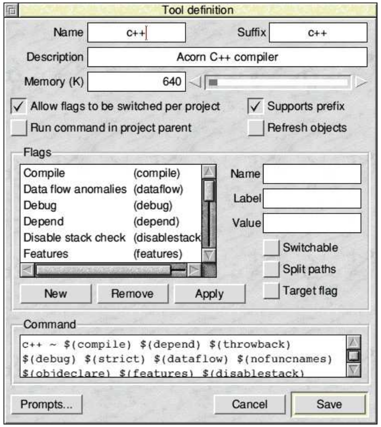
Tools

Tools are the individual commands that are used to build a project.



c++	Acorn C++ compiler
cc	Acorn C compiler
gcc	GNU C++ compiler
objasm	Acorn ARM assembler

Choosing *New* from the menu will display the *Tool definition* window:



Name: c++
Suffix: c++
Description: Acorn C++ compiler
Memory (K): 640

✓ Allow flags to be switched per project
✓ Supports prefix
☐ Run command in project parent
☐ Refresh objects

Flags:

Compile	(compile)
Data flow anomalies	(dataflow)
Debug	(debug)
Depend	(depend)
Disable stack check	(disablestack
Features	(features)

Name:
Label:
Value:
☐ Switchable
☐ Split paths
☐ Target flag

Command:

```
c++ ~ $(compile) $(depend) $(throwback)
$(debug) $(strict) $(dataflow) $(nofuncnames)
$(objdeclare) $(features) $(disablestack)
```

Name
The name of the tool. Normally the name of the command line utility that will be invoked.

Suffix
The suffix that is to be associated with the tool. A tool does not have to have a suffix defined.

Description
A short description of the tool.

Memory
The initial size of the wimpslot to be allocated when running the tool.

Allow flags to be switched per project
If this is switched on then *Project flags* can be defined on a per project basis for the tool. Otherwise system wide settings are used.

Supports prefix
If this is switched on it specifies that the tool supports the use of the *Prefix* command supplied with the Acorn C/C++ suite. If this is not switched on the a *Dir* command will be issued instead.

Run command in project parent
Normally when a tool command is run it is run from within the directory where the project details are stored. Some commands need to be run in the directory that contains the project directory. Setting this flag will switch on this behaviour.

Refresh objects
Some commands will affect the size or permissions of a file. This needs to be reflected in the source or resource window. Setting this flag will cause the details displayed in the window to be refreshed once the command has run.

Flags

This sections allows the individual command line flags for the tool to be defined. Clicking *New* will add a new flag, *Remove* will delete any highlighted flags and *Apply* will store any changes made to the flag.

Name
The name of the flag that will appear in the *Project flags* window.

Label
The label that is used in the *Command* section when defining the tool command.

Value
The flag expected by the command line tool to specify the action being defined. Some special tokens can be embedded in here:

$(includepath)
Will be expanded to list all the defined include paths for the project.

$(librarypath)
Will be expanded to list all the defined libraries for the project.

$(objectlist)
Will be expanded to list all the object files for the project.

$(filecontents)
Will be expanded to list the contents of the file being compiled.

Switchable
If this is selected the flag will appear within the *Project tool flag settings* window.

Split paths
The Value field can contain some special tokens to specify any defined include paths or libraries. For example:

```
-I $(includepath)
```

If this is set then the command will be expanded as follows:

```
-I path1 -I path2 -I path3
```

if it is not set then the command would look like this:

```
-I path1,path2,path3
```

Target flag
Setting this specifies that the flag is to be used as a build target flag. The only affect this will have in practice is to append a (T) to the name of the flag in the *Project flags* window.

Command
The command that is to be run when the tool is invoked. This is made up of references to the defined flags as well as some special tokens:

$(file)
The name of the code being compiled.

$(project)
The name of the project.

$(localpath)
The path of a resource file relative to the main project directory.

$(unixlocalpath)
The path of a resource file relative to the main project directory specified using Unix style naming.

$(suffix)
The suffix of the file being processed. A directory separator of '.' is automatically added if the suffix is present.

$(unixsuffix)
The suffix of the file being processed. If this falls at the end of a path then it will be prefixed with '.'. If it falls within a path then it will be added with a '/' after it.

$(outputsuffix)
This is used when building multiple targets for a project. The suffix that is defined in the Target definition window will be output.

$(temp)
A temporary filename that is guaranteed to be unique for the lifetime of the *Tool Sequence* being run. This allows temporary output from *Tools* that are chained together to be passed from tool to tool.

$(pattern)
The pattern specified in the *Find* window.

$(findpaths)
The path that will be searched when a find operation is triggered.

$(misc)
The *Miscellaneous* flags defined in the *Project tool flag settings* window.

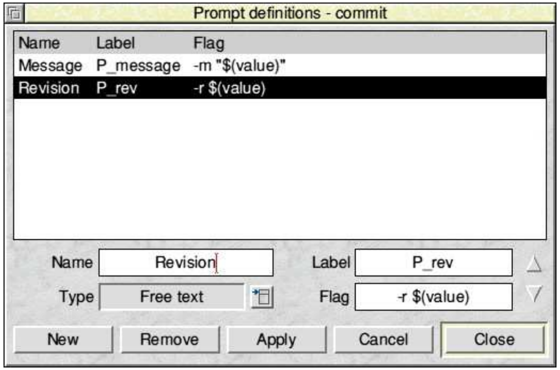
$(viafile)
A temporary file that will contain all the object files and libraries for the project.

The tilde (~) character also has a special meaning. Any text that appears between two tilde characters will be expanded and used as the flag settings for a tool when a makefile is generated. Two ~ symbols together will result in a single ~ symbol in the generated command.

Constants and Prompts (see below) can also be specified in the command.

Prompts

Some commands that are run require the user to specify certain settings at runtime. This can be achieved by defining prompts for a *Tool*. *Prompts* will be displayed before the *Tool* is run to enable the user to set flags and pass other information to the Tool. The *Prompt* window is accessed via the *Prompts* button on the *Tool definition* window.



Name	Label	Flag
Message	P_message	-m "$(value)"
Revision	P_rev	-r $(value)

Name	Revision	Label	P_rev
Type	Free text	Flag	-r $(value)

A highlighted prompt can have the following settings amended:

Name
The name of the *Prompt* that will appear when the *Tool* prompts the user for input.

Label
The label that is inserted in the *Tool* command to be replaced with the value entered at the prompt. *Prompt* labels should have a prefix of P_ to specify that they are prompts.

Type
Prompts can be either *Free text* which require a value to be entered or *On / Off* which will display an option button which the user can switch on or off.

Flag
The full text that will replace the label in the *Tool* command. The *$(value)* variable will be replaced by whatever the user entered. If the prompt is a *Free text* and no value is entered then the label will be removed completely from the *Tool* command.

The following buttons are available at the bottom of the window:

New
Adds a new *Prompt* to the list of prompts for the *Tool*.

Remove
Removes the currently highlighted *Prompt* from the list.

Apply
Applies any changes made to the currently highlighted *Prompt*.

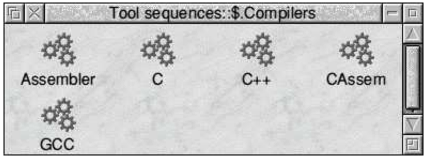
Cancel
Closes the window without storing any changes.

Close
Closes the window and stores the changes for the *Tool*. Note that the changes will not be saved until the *Save* button in the *Tool definition* window is clicked.

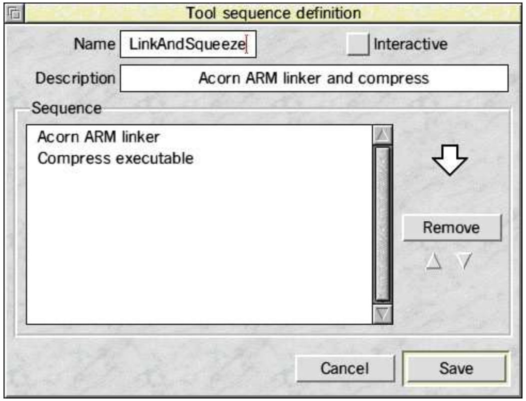
The order of the *Prompts* can be changed by highlighting a *Prompt* and clicking the up or down buttons.

Tool sequences

Tool sequences are chains of tools that are used to build a project.



Choosing *New* from the *Tool sequences* menu will display the *Tool sequence definition* window:



Name: LinkAndSqueeze

Description: Acorn ARM linker and compress

Acorn ARM linker
Compress executable

Name
The name of the sequence.

Description
A short description of what the sequence does.

Interactive
If this is switched on the sequence will not be run using a Task Window. It will be started as a normal WIMP application.

Sequence
The sequence definition. To add to this list simply drag tools from the *Tools* window. A tool can be removed from the sequence by highlighting it and clicking *Remove*.

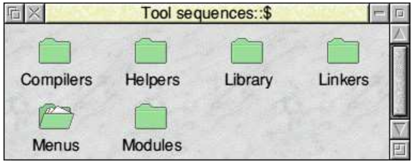
The order of the sequence can be changed using the up and down arrows.

Click *Save* to save the sequence.

Menu sequences

Through *Tool* and *Tool Sequence* definitions *Sourcery* allows the user to run any command line based utility. To further make use of this facility it is possible to define the menu items that appear on the main menu *Project* sub menu and the *Item* sub menus for source code and resources.

This is achieved by creating *Tool Sequence* definitions in the *Menus* sequence directory:



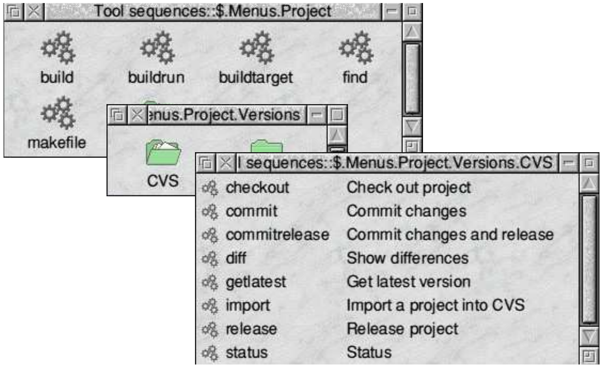
This contains three sub directories:

Project
Any *Tool Sequences* defined in here will appear on the *Project* sub menu from the main menu.

Resources
Tool Sequences defined in here appear on the Item sub menu when it is opened over the *Resources* window.

Source
Tool Sequences defined in here appear on the Item sub menu when it is opened over the *Source* window.

When defining *Tool Sequences* in these windows the user can also define sub directories to group related functions together. The sequences in these directories appear on their own separate sub menus.


Tool sequences::$.Menus.Project
build
buildrun
buildtarget
find
makefile
enus.Project.Versions
CVS
l sequences::$.Menus.Project.Versions.CVS

checkout	Check out project
commit	Commit changes
commitrelease	Commit changes and release
diff	Show differences
getlatest	Get latest version
import	Import a project into CVS
release	Release project
status	Status

There are a number of special *Tool Sequence* names that can be used without having to specify any *Tools* in the sequence. These access internal functions of *Sourcery* without using external command line tools.

compile
Compiles the currently highlighted source code.

build
Builds the project, compiling any source code as necessary.

buildrun
Builds the project, compiling any source code as necessary and launches the project once it has been successfully built.

buildtarget
Displays a window which allows the selection of multiple targets to build.

find
Runs the find tool to search source code and resources.

makefile
Generates a makefile for the project.

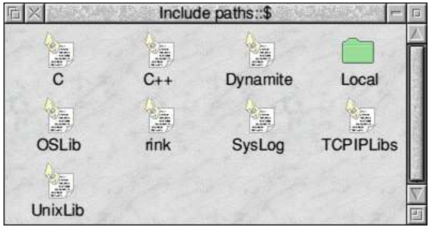
open
Opens any highlighted source code or resources in the relevant editor.

openparentdir
Opens the filer window that contains the current highlighted project.

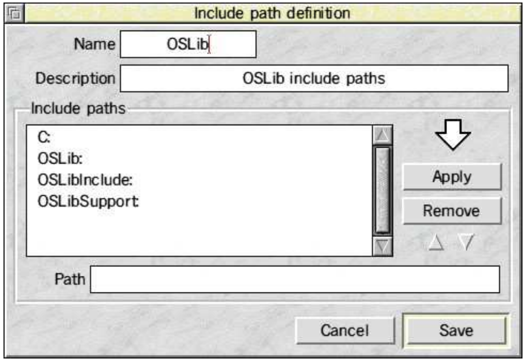
touch
Touches any highlighted source code.

Include paths

Includes paths specify where to look when compiling code to find header files:



To add a new path choose *New* from the menu to display the *Include path definition* window.



Name: OSLib

Description: OSLib include paths

Include paths:

C:
OSLib:
OSLibInclude:
OSLibSupport:

Path:

Name
The name of the include path definition.

Description
A short description of the include path definition.

Include paths
The paths that make up the definition. Filer directories containing header files should be dragged to this window. One definition can contain multiple paths as shown in the example.

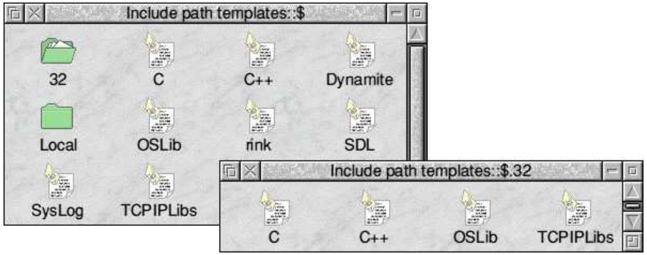
Clicking on a path will highlight it and populate the *Path* field. This allows the include path to be further refined. Click *Apply* to store the changes.

A path can be removed by highlighting it can clicking *Remove*.

The order of the paths can be changed though the use of the up and down arrows.

Include paths and Targets

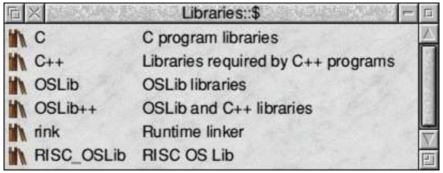
Sourcery can be set up to automatically use different include path definitions for different *Target* builds. Creating a new directory with the name of the *Target* suffix (see later) will cause *Sourcery* to look in that directory first when looking up Include path definitions. In the following image default settings for C, C++ and other include paths are defined but additionally there is a sub directory with additional entries for 32 bit specific targets.



Sourcery will automatically use the 32 bit paths when building 32 bit code and only fall back on the defaults if a 32 bit variant cannot be found.

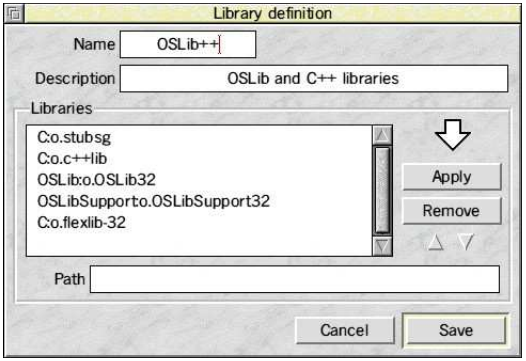
Libraries

Libraries specify the external functionality that will be linked into a project when it is built:



C	C program libraries
C++	Libraries required by C++ programs
OSLib	OSLib libraries
OSLib++	OSLib and C++ libraries
rink	Runtime linker
RISC_OSLib	RISC OS Lib

To add a new library choose *New* from the menu to display the *Library definition* window.



Name: OSLib++

Description: OSLib and C++ libraries

```
C:o.stubsg
C:o.c++lib
OSLib:o.OSLib32
OSLibSupport:o.OSLibSupport32
C:o.flexlib-32
```

Path:

Name
The name of the library definition.

Description
A short description of the library definition.

Libraries
The libraries that make up the definition. Library files should be dragged to this window. One definition can contain multiple libraries as shown in the example.

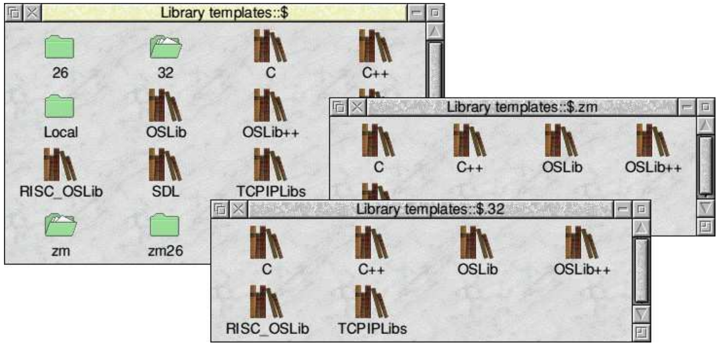
Clicking on a library will highlight it and populate the *Path* field. This allows the library to be further refined. Click *Apply* to store the changes.

A library can be removed by highlighting it can clicking *Remove*.

The order of the libraries can be changed though the use of the up and down arrows.

Libraries and Targets

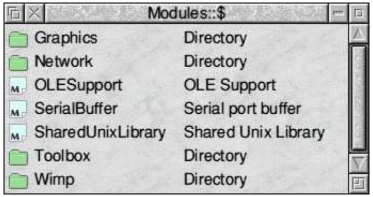
Sourcery can be set up to automatically use different library definitions for different *Target* builds. Creating a new directory with the name of the *Target* suffix will cause *Sourcery* to look in that directory first when looking up Library definitions. In the following image default settings for C, C++ and other libraries are defined but additionally there are a number of sub directories with additional entries for module,26 bit and 32 bit specific targets.



Sourcery will automatically use the correct libraries when building different targets and only fall back on the defaults if a suitable variant cannot be found.

Modules

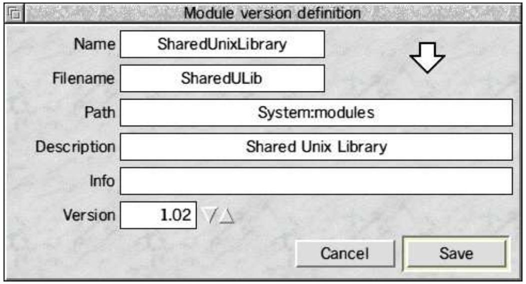
This window allows modules definitions to be set up to allow the easy maintenance of module references in !Run files.



Modules::$

Graphics	Directory
Network	Directory
OLESupport	OLE Support
SerialBuffer	Serial port buffer
SharedUnixLibrary	Shared Unix Library
Toolbox	Directory
Wimp	Directory

A new module can be added by choosing *New* from the menu to display the *Module version definition* window:



Module version definition

Name	SharedUnixLibrary
Filename	SharedULib
Path	System:modules
Description	Shared Unix Library
Info	
Version	1.02

Name

The name of the module. This is the name that is returned when issuing a *Help command, not the file name.

Filename

The filename of the module as it appears on disc.

Path
The path in System where the module resides.

Description
A short description of what the module does.

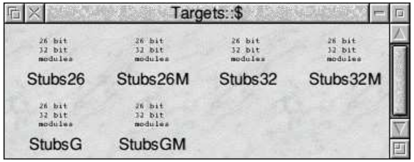
Info
Extra information about the module, such as where it can be downloaded from, that can be displayed as part of the error when the module is not found on the users system

Version
The latest version number of the module.

Click *Save* to save the module definition.

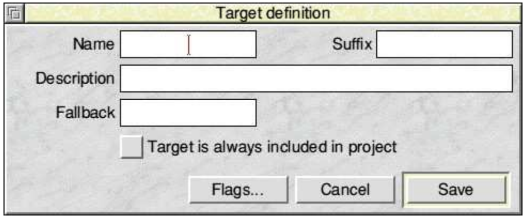
Targets

This window allows the targets available when building a project to be configured:



Targets are primarily provided to allow different versions of a library to be built from the same source code. This is especially useful if a library needs to be used with a normal application and with a module. Module code must be compiled with a different set of compiler options.

A new target can be added by choosing *New* from the menu to display the *Target definition* window:



Name

The name of the target. This must not contain spaces.

Suffix
The suffix that will be appended to any object code names that are produced when building this target.

Description
A description of what the target is.

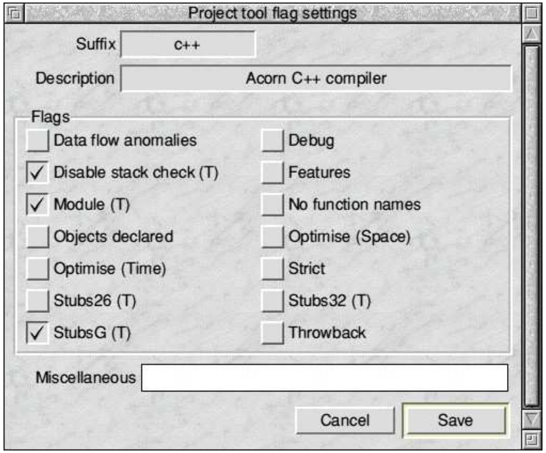
Fallback
This is used when *Include path* or *Library* definitions do not exist for the *Target* but the default definitions should not be used. For example, consider the situation where *Include path* definitions exist for 32 bit libraries. When building 32 bit targets these will be used. However, a 32 bit module target might also exist which should use these paths. Setting the *Fallback* suffix for this target to the same as the suffix for the 32 bit target avoids the need to set up specific 32 bit module *Include path* definitions which would be the same as the ordinary 32 bit definitions.

Target is always included in project
If this is set the target will always be built for a project and cannot be excluded.

Click *Save* to save the target definition.

Target flags

Different targets are generated by specifying different compiler flags. Flags are defined in the same way as *Project tool* flags.


Project tool flag settings

Suffix: c++

Description: Acorn C++ compiler

Flags

- [ ] Data flow anomalies
- [✓] Disable stack check (T)
- [✓] Module (T)
- [ ] Objects declared
- [ ] Optimise (Time)
- [ ] Stubs26 (T)
- [✓] StubsG (T)
- [ ] Debug
- [ ] Features
- [ ] No function names
- [ ] Optimise (Space)
- [ ] Strict
- [ ] Stubs32 (T)
- [ ] Throwback

Miscellaneous:

Unlike *Project tool* flags however, only those flags which have a (T) after the name should be specified. These will then override the settings at project level for these flags.

Sourcery allows tool flags to be specified in three different places and the relationship between the different flag settings can be confusing.

Project type flags

These are the default flag settings for each different tool within a project type. Only non target flags (those without (T)) in their name) should be set at this level. When a new project is created based on a project type, the tool flags for the project type are copied to the project. Changing the default project type settings will then have no effect on the locally copied project flags.

Project flags
The local copy of tool flags based on the settings for the type of project. These can be modified and changed without affecting any other projects. Like *Project type flags,* you should avoid setting those flags with (T) in their name although advanced users can set these if they want.

Target flags
The flags that apply to a particular target type. Only those flags with (T) in their name should be set at this level. Any flags that are set will be merged with the *Project flags* when a piece of code is compiled. Thus picking a different target will cause a different set of *Target flags* to be merged with the *Project flags* at compile time.

The table below shows the relationship between the different flags.

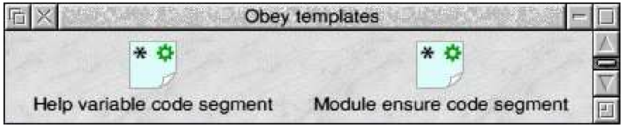
Project flags		Target flags		Compile time flags
Optimise		Optimise		Optimise
Off		Off		Off
Debug		Debug		Debug
On		Off		On
Strict	+	Strict	=	Strict
On		Off		On
Module (T)		Module (T)		Module (T)
Off		On		On
26 bit (T)		26 bit (T)		26 bit (T)
On		Off		On

The final flag, *26 bit (T)*, is an example of how not to set flags. Because this is a target (T) flag it should not be set at project level.

Note that a compiler or tool must be capable of producing code for a particular type of target. *Sourcery* will not prevent a tool from being used to generate object code for a target if that tool does not have the required functionality, and will not magically endow a tool with the ability to generate code of a certain type.

Obey templates

Sourcery allows the automatic creation of sections of the !Run and !Boot files that deal with the loading of any modules required by an application and the generation of Help variables. *Obey templates* allows the segments of code that make up these sections to be edited and customised. Clicking on this displays the *Obey templates* window:



Double clicking on an icon will display the relevant code.

Help variable code segment

This is responsible for setting up the entries in the !Run and !Boot files that provide simple version and help information.

By default the default for this segment looks as follows:

```
Set ~application~$Version "~version~"
Set ~application~$Web "~website~"
Set ~application~$Title "~title~"
Set ~application~$Publisher "~publisher~"
Set ~application~$Description "~description~"
```

There are a number of special tokens that will be replaced when the segment is used. The values of these are all taken from the corresponding entries in the *Project definition* window:

~application~ & ~appname~
The name of the application taken from the *Project definition* window without a prefixing ! if present.

~description~
A brief description of what the application does.

~publisher~
The name of the company or individual publishing or distributing the application.

~title~
The title of the application.

~version~
The version number of the application.

~website~
The website where the application or resources for the application can be found.

Module ensure code segment

This segment loads a module within the !Run file, stating which version is needed and potentially some helpful information if it cannot be found.

This looks as follows:

```
     RMEnsure ~name~ 0.00 RMLoad ~path~.~filename~
     RMEnsure ~name~ ~version~ Error You need ~name~ ~version~ or later to run
          ~appname~.~info~
```

Again there are a number of special tokens that are replaced when the segment is used:

~application~ & ~appname~
The name of the application taken from the *Project definition* window without a prefixing ! if present.

~filename~
The filename of the module. This is taken from the *Module version definition* window.

~info~
A brief description of where the module can be found. This is taken from the *Module version definition* window.

~name~
The name of the module. This is taken from the *Module version definition* window.

~path~
The path where the module is located. This is taken from the *Module version definition* window.

~version~

The version of the module. This is taken from the *Module version definition* window or the *Applications modules* window.

Module version ensure code segment

Sometimes an application requires different versions of the same module for different version of the operating system. This normally only relates to Toolbox modules. This segment enables the !Run file to execute a different check for different versions of the operating system.

This looks as follows:

```
If ((”<Boot$OSVersion>” LEFT 1 = ~version~) AND (”<~application~$RunComplete>” = “”))
     Then Run <~application~$Dir>.!RunFiles.RunOS~version~
If “<~application~$RunError>” <> “” Then Error “Error loading modules”
```

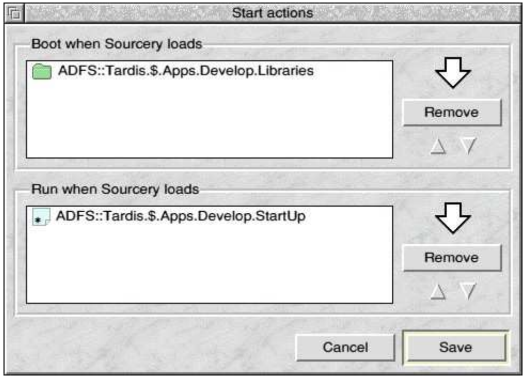
Again there are a number of special tokens that are replaced when the segment is used:

~application~ & ~appname~
The name of the application taken from the *Project definition* window without a prefixing ! if present.

~version~
The version of the operating system that the file should be run for. This is only ever a major version like 3, 4, 5 or 6.

Start actions

Start Actions allow other applications to be booted or run automatically when *Sourcery* is loaded:



The window comprises of two areas: Applications or paths to be booted when *Sourcery* is loaded and applications or paths to be run when *Sourcery* is loaded. Both sections work in the same way:

Adding a new path

Simply drag an item from a filer window to the large arrow corresponding to the section to which the path should be added. Any filer object can be dragged to this window.

List area

The list area shows a list of applications or paths that will be booted or run. Clicking on a path will highlight it.

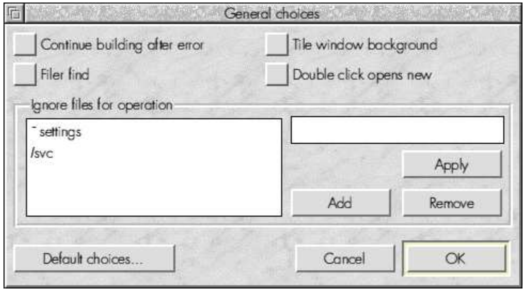
Remove
Clicking *Remove* whilst a path is highlighted will cause it to be removed from the list.

Up and Down
Clicking the *Up* or *Down* arrows while a path is highlighted will move it up or down the list. This allows the order in which the paths are processed to be controlled.

When the required paths have been added to the window click *Save* to store the choices or *Cancel* to close the window without making any alterations.

General

General choices contain any miscellaneous choices that do not fit elsewhere.



~settings
/svc

Continue building after error
Normally when building a project, if a piece of code fails to compile the process will abort. Switching this only will enable the process to continue and compile other code even when an error is encountered.

Filer find
RISC OS Select allows the Filer to support plugins for certain operations. The excellent !Locate utility can act as a replacement for the Filer find action. If this is switched on *Sourcery* will use Filer action to when Find is clicked rather than the inbuilt find functionality,

Tile window background
One of the strengths of *Sourcery* is the consistent, filer like interface. However, sometimes when a large number of filer and *Sourcery* windows are open at the same time it can become a little difficult to keep track of which is which. If this is switched on then all *Sourcery* windows will have a different background to make it easier to identify them.

Double click opens new
If this is switched on then double clicking in a window will have the same effect as choosing *New item...* from the *New* menu.

Ignore files for operation
A number of options in *Sourcery* work on all the files in a project, or part of a project. For example, source control can be carried out on the whole project.

There are occasions when files exist within project directories which you would not want to be included in these operations. For example, *Subversion* creates */svc*.

Files likes this which should be ignored can be defined here.

Default choices...
Clicking this button will prompt you to confirm that any changes that have been made to compiler and tool definitions should be replaced with the defaults. Any new definitions that have been created will be unaffected.

Toolbar

Along the top of some windows is a toolbar. This provides a quick way of performing some common operations.

Build

Clicking this icon will try and build the project. Any source code that needs compiling will be compiled and then combined together into the target executable or library.

Run

The *Run* icon performs the same action as the *Build* icon but for applications will launch the application after it has been successfully built.

Compile

Sometimes it is necessary to only compile certain pieces of code without building the whole project. Selecting those pieces of code and clicking this icon will just compile the selected code without building it.

Open

Selecting multiple pieces of code and clicking this icon will open the code in a text editor.

New

Clicking the *New* icon will open a window allowing a new item to be created. This is the same as choosing *New* from the menu.

Touch

Sometimes it is necessary to trigger the recompilation of code without having made any changes to it. This is often referred to as 'touching' the code. Select the code to be touched and click this icon.

Find

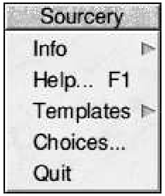
Clicking Find will open a window that allows the source code to be searched for an expression.

Templates

Clicking this will display the relevant Templates window.

The iconbar menu

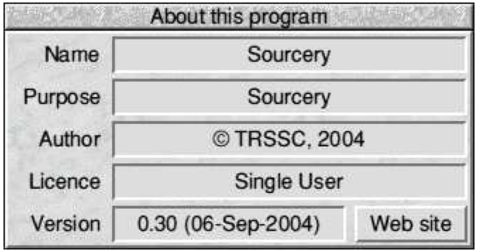
Clicking Menu over the *Sourcery* icon on the iconbar will display this menu:



The following items are available:

Info

Moving the mouse pointer over this item displays the *About this program* window which displays version information about *Sourcery*.


About this program

Name	Sourcery	
Purpose	Sourcery	
Author	© TRSSC, 2004	
Licence	Single User	
Version	0.30 (06-Sep-2004)	Web site

Help

Choosing this item will display the help supplied with the application.

Choices
Choosing this will display the *Choices* window which is discussed in the *Choices* chapter.

Quit
Choosing this item will cause *Sourcer*y to exit.

Appendix A

This appendix contains details of where some useful tools, libraries and information can be found.

Name	Description	Link
Acorn C/C++	The C/C++ compiler suite originally from Acorn. Now developed by Castle.	www.iyonix.com
GCCSDK	GCC compiler and tools. A free alternative to the above.	gccsdk.riscos.info
UnixLib	Attempts to provides a full implementation of all Unix library calls. The default library for GCC.	gccsdk.riscos.info
OSLib	An API library giving easy, efficient and type safe access to all RISC OS SWI calls.	ro-oslib.sourceforge.net
DeskLib	A library to aid in the creation of RISC OS applications.	www.riscos.info/desklib
DeskDebug	A desktop debugging tool.	buyit.spellings.net
RISCOS.Info	A useful resource when programming on RISC OS.	www.riscos.info