

Getting Started with the Catalina Geany IDE

Table of Contents

Introduction.....	2
Using Geany to build and run a program from scratch.....	3
Using Geany to build and run an existing simple program.....	13
Using Geany to build and debug a complex program.....	21
Using Geany to build and run a program using XMM RAM.....	27
Additions to the Geany IDE for Catalina.....	31
Catalina Project Properties.....	31
Catalina Build Commands.....	33
The Geany Build Menu.....	35
The C commands.....	35
Independent commands.....	35
Execute commands.....	36
An example Makefile.....	38
Using make from the Catalina Command Line.....	41
Using the Supplied Catalina Geany Project Files.....	42

Introduction

Catalina has an Integrated Development Environment (IDE), supported on both Windows and Linux. It is a customized version of the "Geany" IDE¹. Geany is a very "lightweight" IDE, and is very simple to use. It is perfect for the types of C programs typically written for the Propeller.

This document introduces the new IDE, and gives some examples of how to use it:

- Using Geany to build and run a program from scratch.
- Using Geany to build and run an existing simple program.
- Using Geany to build and debug a complex program.
- Using Geany to build and run a program using XMM RAM.
- Using Makefiles with Geany.

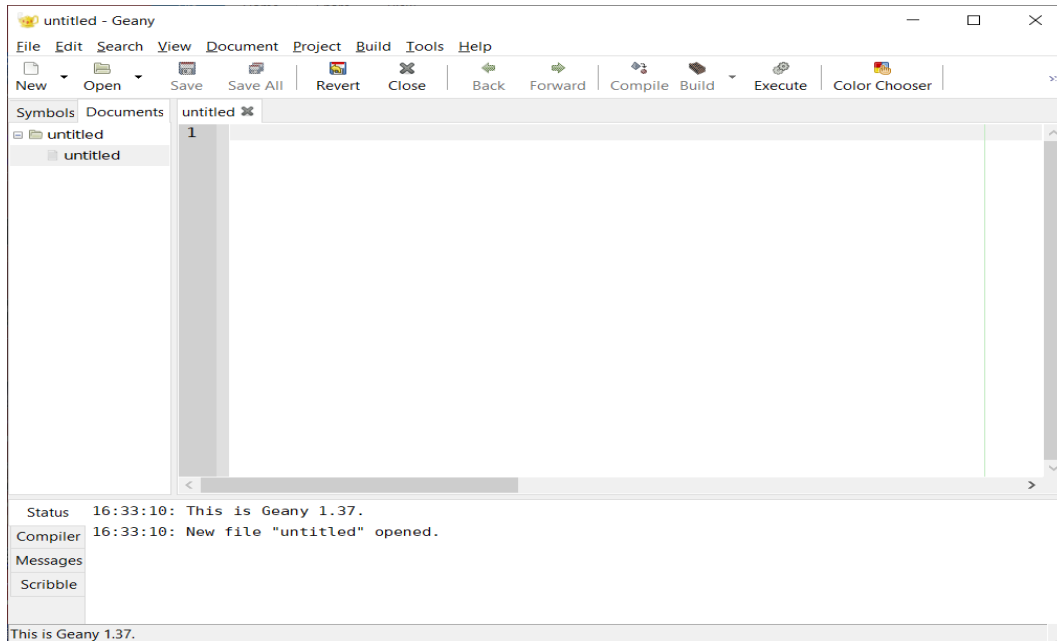
Finally, there is a section that discusses the additions made to the standard Geany IDE specifically for Catalina. This document assumes you are running the Catalina Geany IDE under Windows. There are minor cosmetic differences under Linux, but the functionality is identical.

¹*Don't ask me – I guess the spelling "Genie" was already used by another program!*

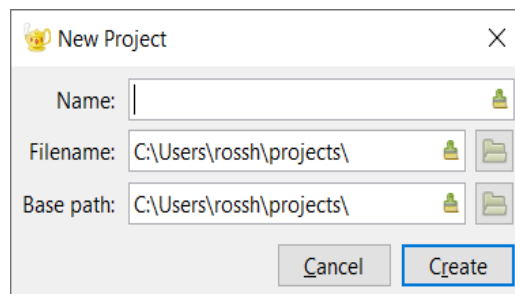
Using Geany to build and run a program from scratch

On Windows, start the **Catalina Geany IDE** from the Start Menu shortcut of that name, or using the command **catalina_geany** from a Catalina Command Line.

On Linux, open a terminal window, and execute the command **catalina_geany** (this command can be found in the “/opt/catalina/bin” directory²). You should see a window similar to this:



The first thing we typically want to do is create a new project. So select the **Project→New ...** menu entry. You should see a dialog like this:

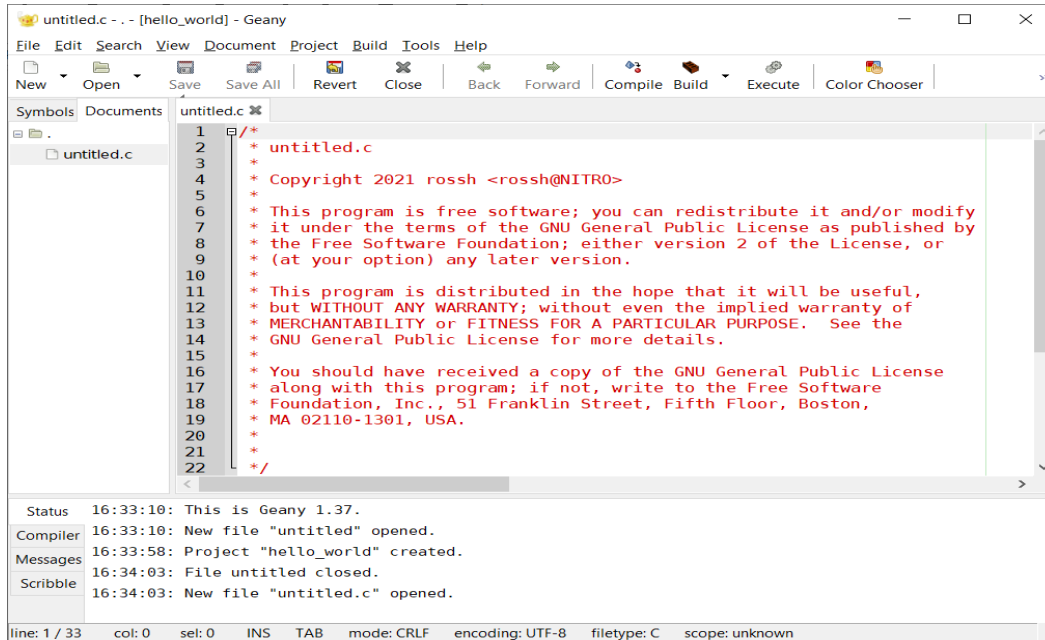


² Note that on Linux, Catalina’s version of Geany expects to be installed in /opt/catalina/catalina_geany, which is where it will end up if you install Catalina to its default location (i.e. /opt/catalina). To install it elsewhere, Geany may need to be recompiled from source.

By default in Windows, all projects are created in your Windows User directory, in a subdirectory called "projects" (on Linux, this will be in a subdirectory of your home directory). Note that if the two paths are not the same at this stage, then the project file will not be put in the project directory.

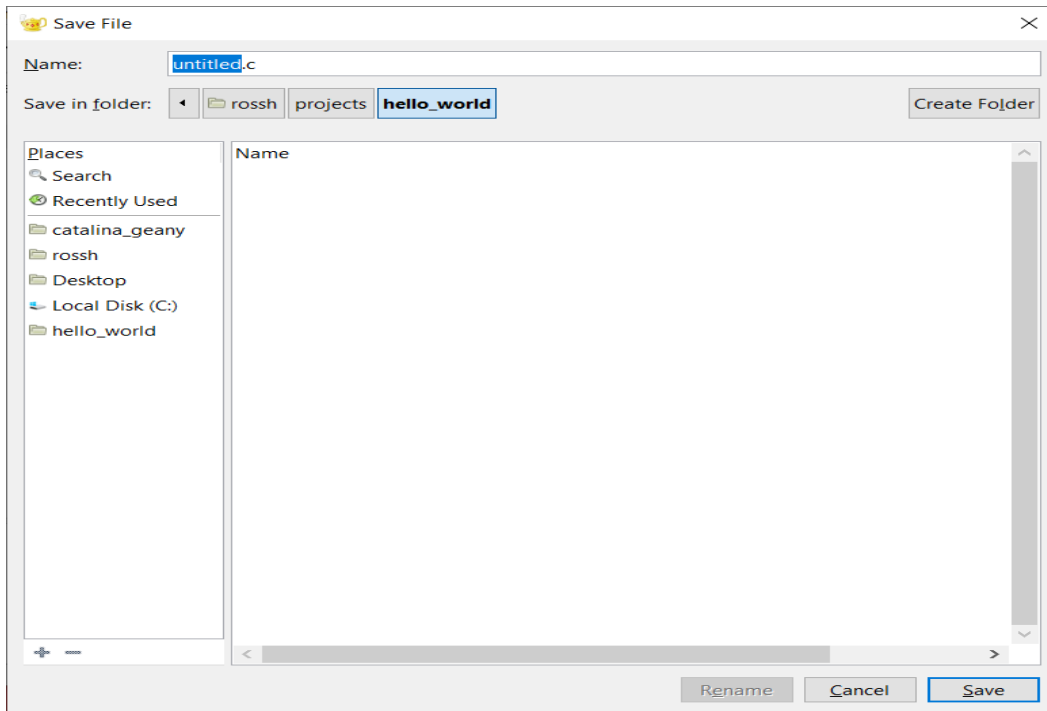
All you need to do here is enter the project name – in this case, enter the name **hello_world** and press **Create**³. You will be prompted to create the project directory. Just press **OK**.

Next we will create a file, using one of Geany's many built-in file templates. To do this, select **File→New (With Template)→main.c**. You will see the new file in a window called *untitled.c* – something like this:



³³ Project and file names may contain spaces, but for reasons that will become apparent later in the section discussing Geany and Makefiles, they are not recommended because they complicate things and may not work correctly in some versions of the make utility.

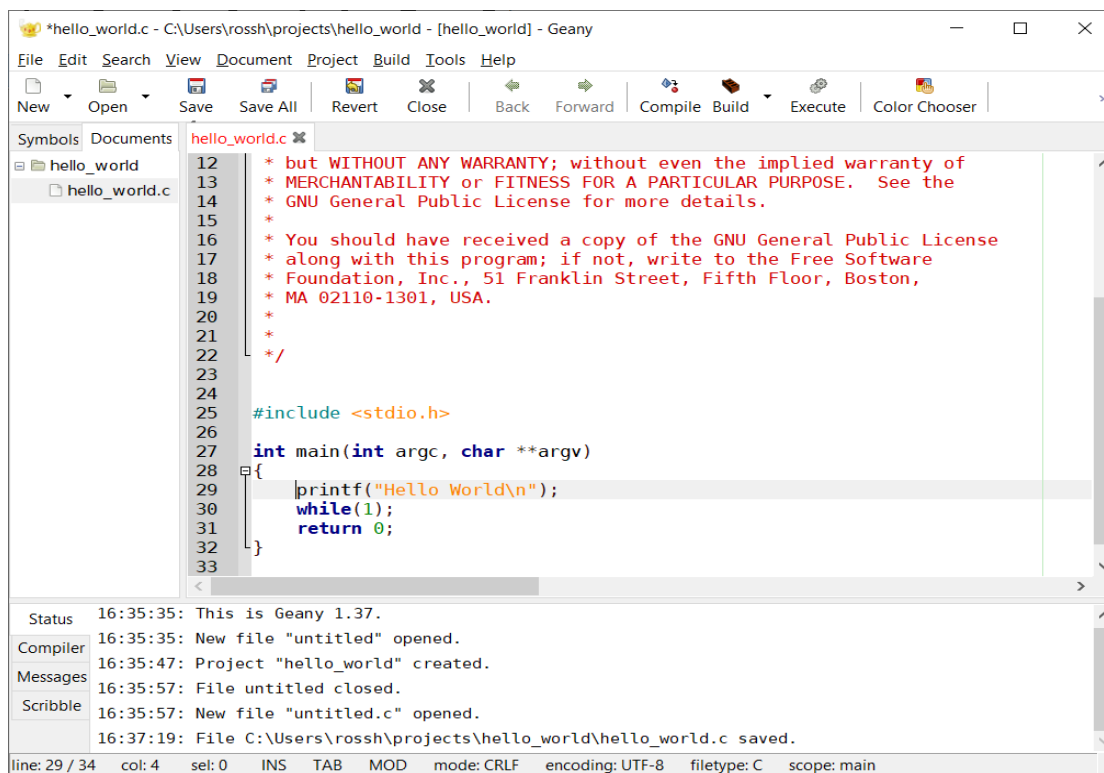
The first thing we want to do is rename and save the file. To do this select **File**→**Save As ...** - the following dialog box will appear. Enter *hello_world.c* as the file name and press **Save**. This file will be saved in your project's directory:



Next, scroll to the bottom of the file in the right hand pane, and enter the following two lines in the main function, just before the line that says "return 0;". Note that Geany will assist you as you type.

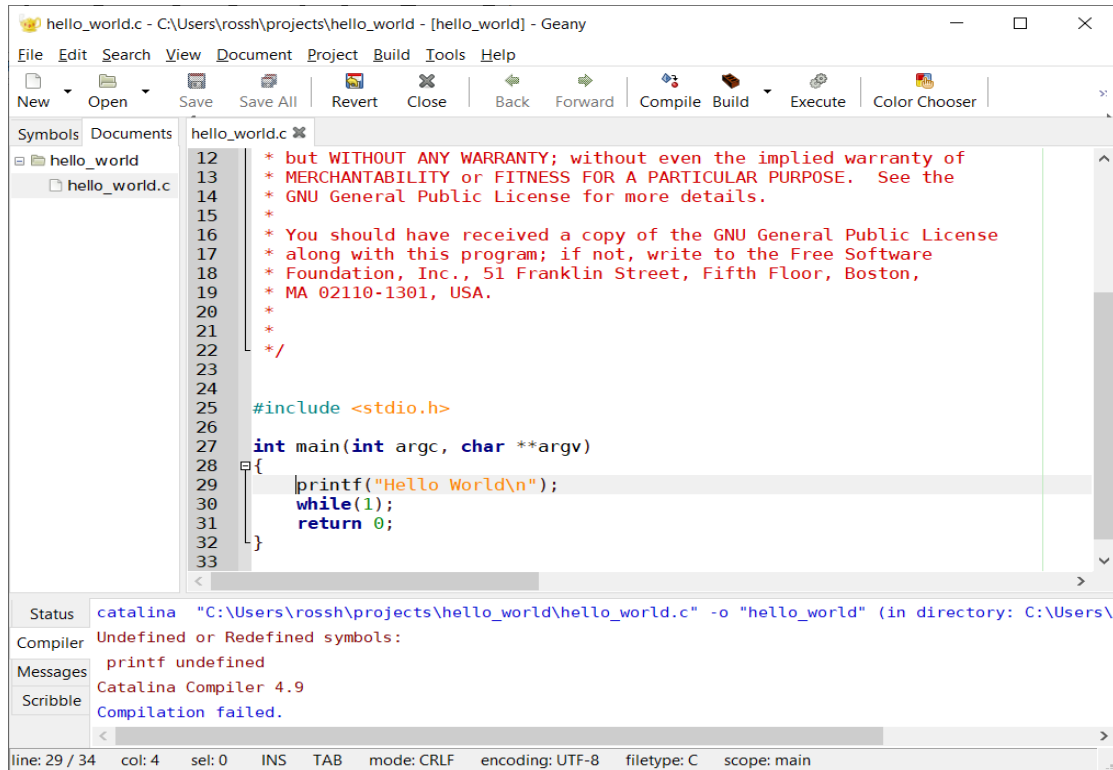
```
printf("Hello World\n");  
while(1);
```

Your window should now look like this:



And now we are ready to try building the project. To do so, select the Build→Build File menu entry, or press the Build button. Note: we will get undefined symbols from this step – this is expected! We will explain why, and fix this error in the next step.

This is what you should see:



```

hello_world.c - C:\Users\rossh\projects\hello_world - [hello_world] - Geany
File Edit Search View Document Project Build Tools Help
New Open Save Save All Revert Close Back Forward Compile Build Execute Color Chooser
Symbols Documents hello_world.c
hello_world
  hello_world.c
12  * but WITHOUT ANY WARRANTY; without even the implied warranty of
13  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
14  * GNU General Public License for more details.
15  *
16  * You should have received a copy of the GNU General Public License
17  * along with this program; if not, write to the Free Software
18  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
19  * MA 02110-1301, USA.
20  *
21  *
22  */
23
24
25 #include <stdio.h>
26
27 int main(int argc, char **argv)
28 {
29     printf("Hello World\n");
30     while(1);
31     return 0;
32 }
33
Status catalina "C:\Users\rossh\projects\hello_world\hello_world.c" -o "hello_world" (in directory: C:\Users\
Compiler Undefined or Redefined symbols:
Messages printf undefined
Scribble Catalina Compiler 4.9
Compilation failed.
line: 29 / 34 col: 4 sel: 0 INS TAB mode: CRLF encoding: UTF-8 filetype: C scope: main

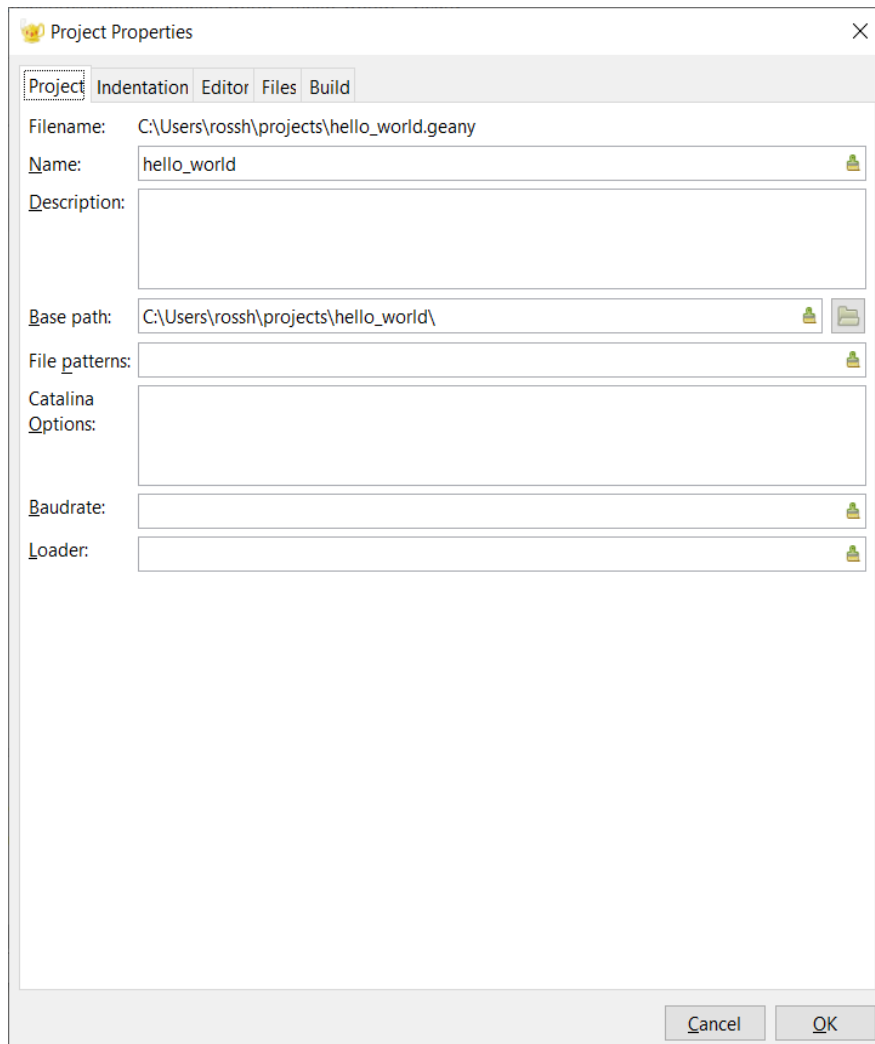
```

The reason the compilation failed is that `printf` is undefined. This is because we have not told Geany which version of the C Library to link with. Catalina has several different versions of the C Libraries you can choose from, and it does not have a default setting.

The reason for this is that the standard C libraries are significantly smaller if you exclude specific things you know you won't need in your program. Catalina has four versions of the standard C libraries:

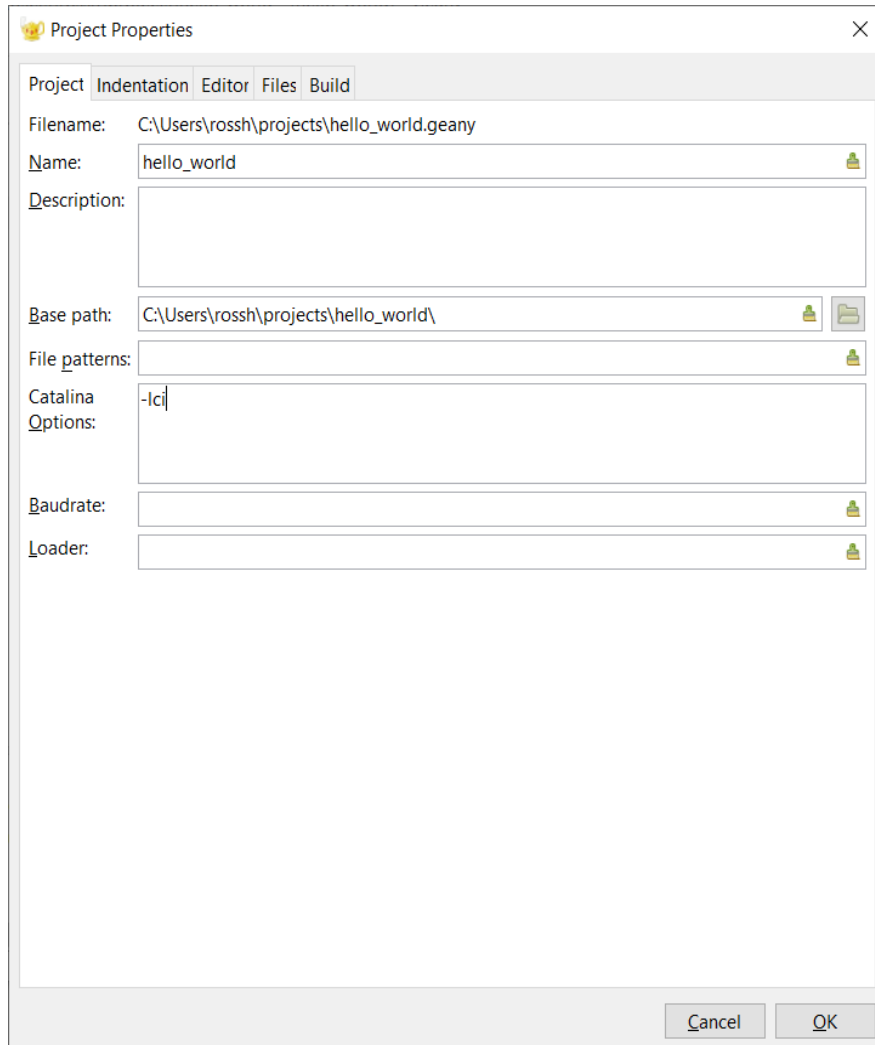
- lci** an *integer* version that does not include floating point I/O in functions such as `printf` (note that you can still *use* floating point – you just cannot perform I/O on them).
- lc** a version that includes floating point I/O in functions such as `printf`.
- lcix** an *extended* version that does not support I/O of floating point, but does include file I/O (on platforms that have an SD Card).
- lcx** an *extended* version that includes floating point I/O, plus file I/O (on platforms that have an SD Card).

To set Catalina-specific project options, such as which version of the C library to use, select the **Project→Properties** menu item. You will see a window like this:



If you are familiar with the standard Geany IDE, you will notice your first differences from standard Geany in the additions to this dialog box. They are the **Catalina Options**, **Baudrate** and **Loader** fields. These are explained in detail later. For now, all we really need to do is add **-lci** to the **Catalina Options** field – this tells Geany we want to link our project with the integer version of the standard C library.

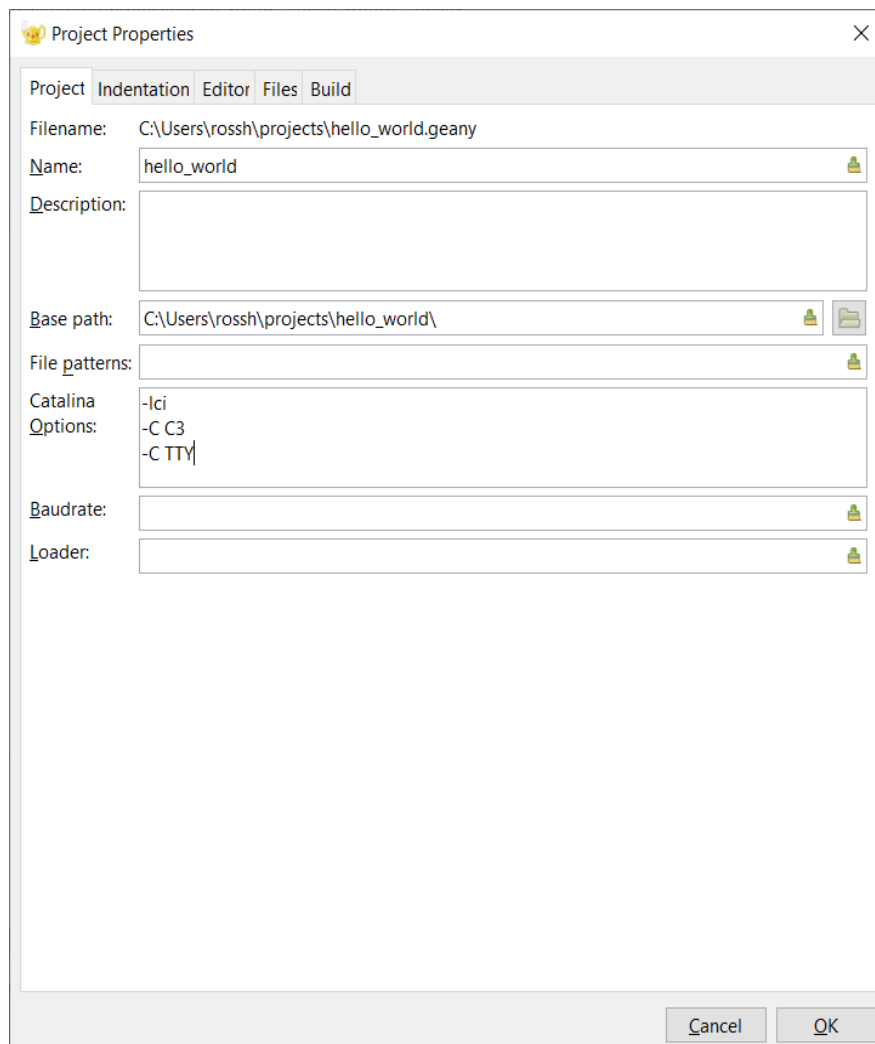
The dialog box should then look like this:



However, while we are adding Catalina options, we can take this opportunity to also add any necessary options for our platform, and also tell the project we want to use the TTY plugin for serial I/O (in case this is not the default). This is not required if you are using a standard Propeller platform that is compatible with Catalina's CUSTOM configuration because this is the default, but let's assume you instead have a C3 board. In that case, you would need to also add **-C C3 -C TTY** to the Catalina options.

NOTE: This tutorial assumes you are using a Propeller 1. If you are using a Propeller 2, you should also add **-p2** in the options field, and either **230400** or **0** in the baud rate field. The baudrate defaults to **115200** if not specified, but specifying it as **0** means to use the default appropriate to the Propeller chip.

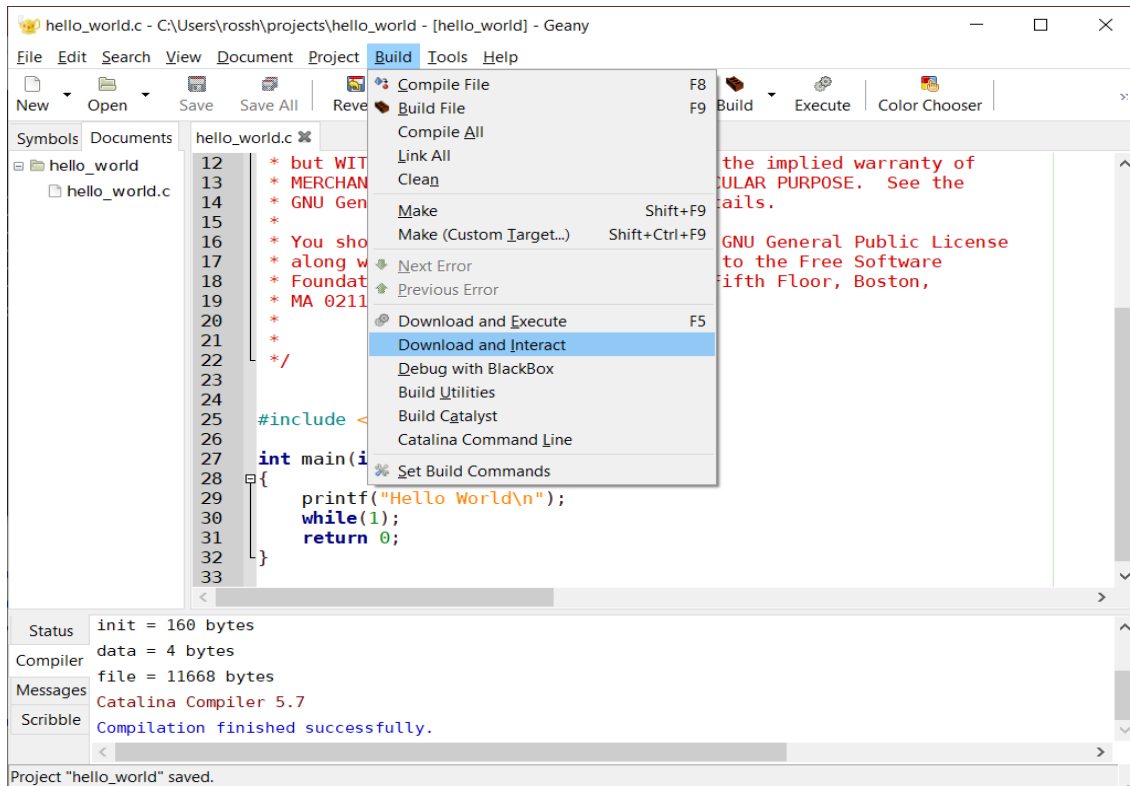
To make the options more readable, you can enter each one on a new line in the **Catalina Options** field. For example:



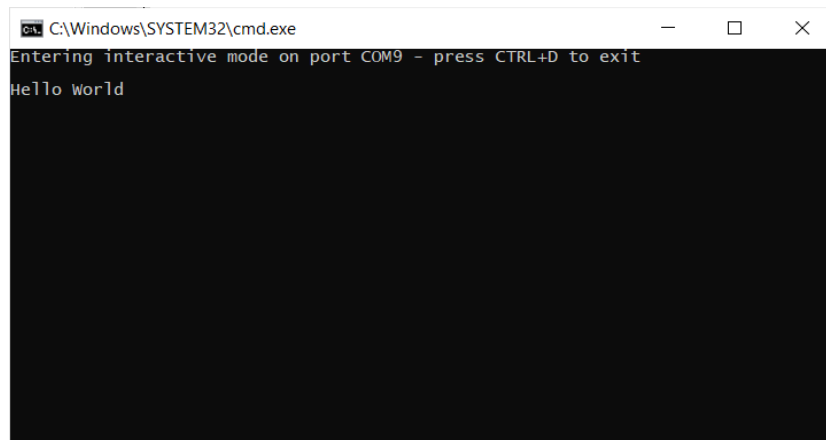
Press **OK** to close the dialog, and then select the **Build**→**Build File** menu item again, or press the **Build** button again.

This time, the compilation should succeed.

Now we can download and execute the program. Because we are using serial I/O as our interface, we want to choose the **Build→Download and Interact** menu item. Otherwise, we would not see any output:



When we do so, we should see an interactive terminal window open up, displaying the output from the program:



For programs that consist of a single C source file, that's all there is to it! You can now close this terminal window.

Just to complete this tutorial, shut down and restart Geany from the **Catalina Geany IDE** menu

item (or the **catalina_geany** command). When you do so, you will notice that Geany always restarts where it left off. Normally, this is what you want to do. If it is not, you should close the open project.

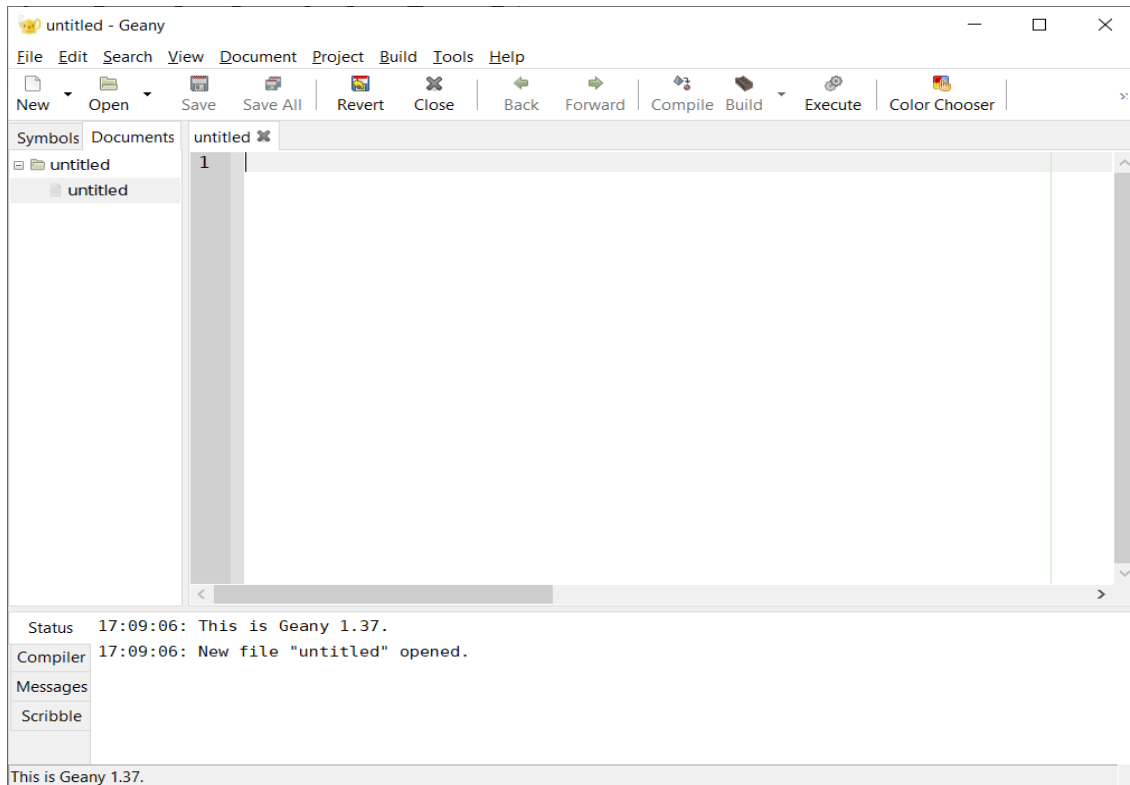
Note that to close the project, you *must* use **Project→Close** menu item. If you just press the **Close** button, it closes the current *file*, but not the *project* – which is not usually what you want to do⁴

⁴ *Geany is a little ambiguous about the state of open files – it is not always obvious whether an open file is part of a project or not. You may find you close the project but some files are left open. Or vice-versa. If you accidentally close a file that is part of a project instead of closing the project, don't panic! The file hasn't disappeared – you just need to re-open the file within the project again using the **File→Open** command while the project itself is open. When you close the project or close Geany, the project will remember all the open files as being part of the project session, and will reopen them when you reopen the project. To make matters more confusing, it is possible to have multiple files open with the same name (but in different directories) at the same time, which may mean you think you are editing a file in the project, but you are editing a completely different file. It is recommended to always close any open files that are **not** part of the project before closing the project or closing Geany.*

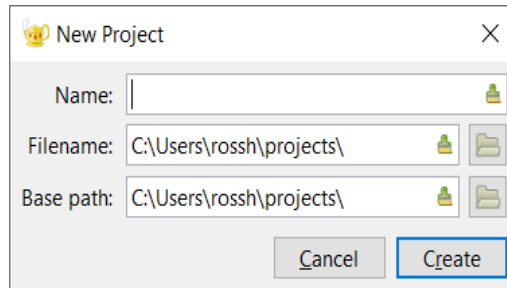
Using Geany to build and run an existing simple program

If Geany is not running, start the **Catalina Geany IDE** from the Start Menu shortcut of that name (on Linux, or from a Catalina Command Line, execute the command **catalina_geany**).

If there is a project open, close it by selecting **Project→Close**, and if there are any C files left open, close them by selecting **File→Close**. You should see a window like this (there may or may not be a file called "untitled" still open – that doesn't matter):

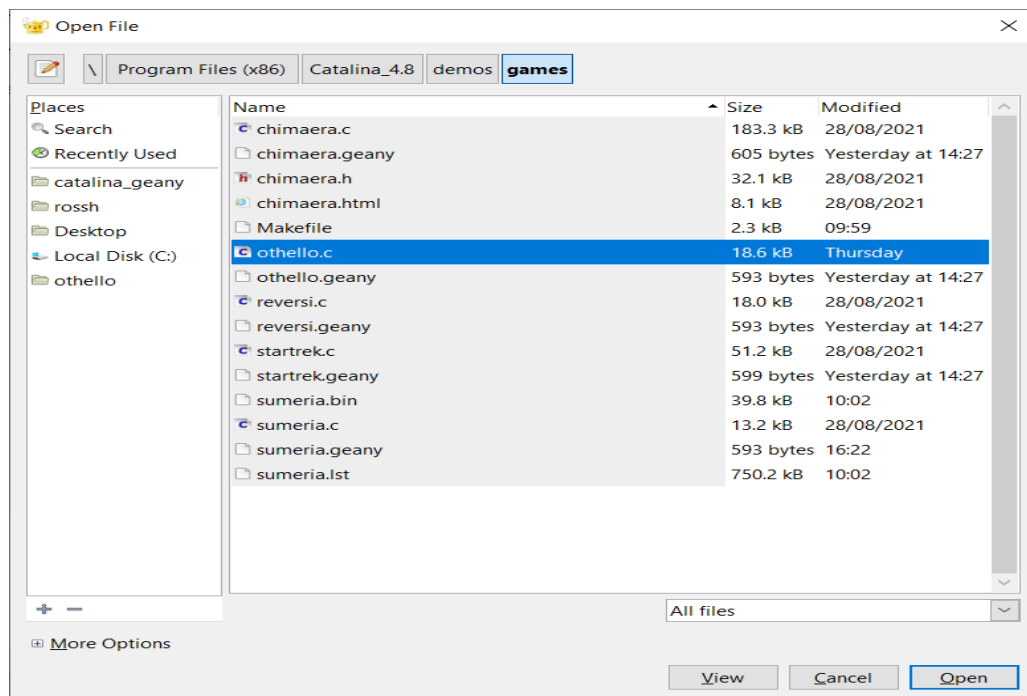


Next, create a new project by selecting the **Project→New ...** menu entry. You should see a dialog like this:

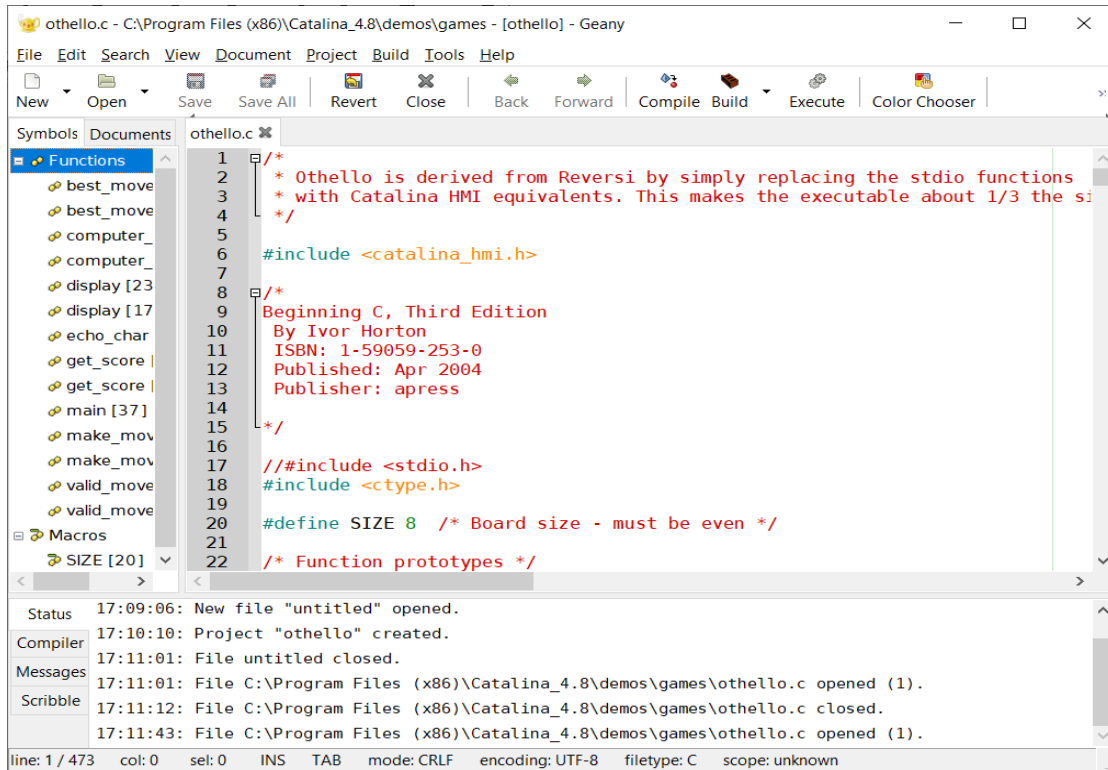


Enter the name **othello** and press **Create**. You will be prompted to create the project directory. Press **OK**.

The next thing we want to do with our new project is include the othello.c program code provided as a Catalina demo program. Select File→Open ... or press the Open button on the Toolbar. You will be presented with an Open File dialog. Navigate to the demos\games directory where Catalina is installed (typically, C:\Program Files (x86)\Catalina\demos\games on Windows, or /opt/catalina/demos/games on Linux) and select the file othello.c from this directory, as shown below. Then press Open:



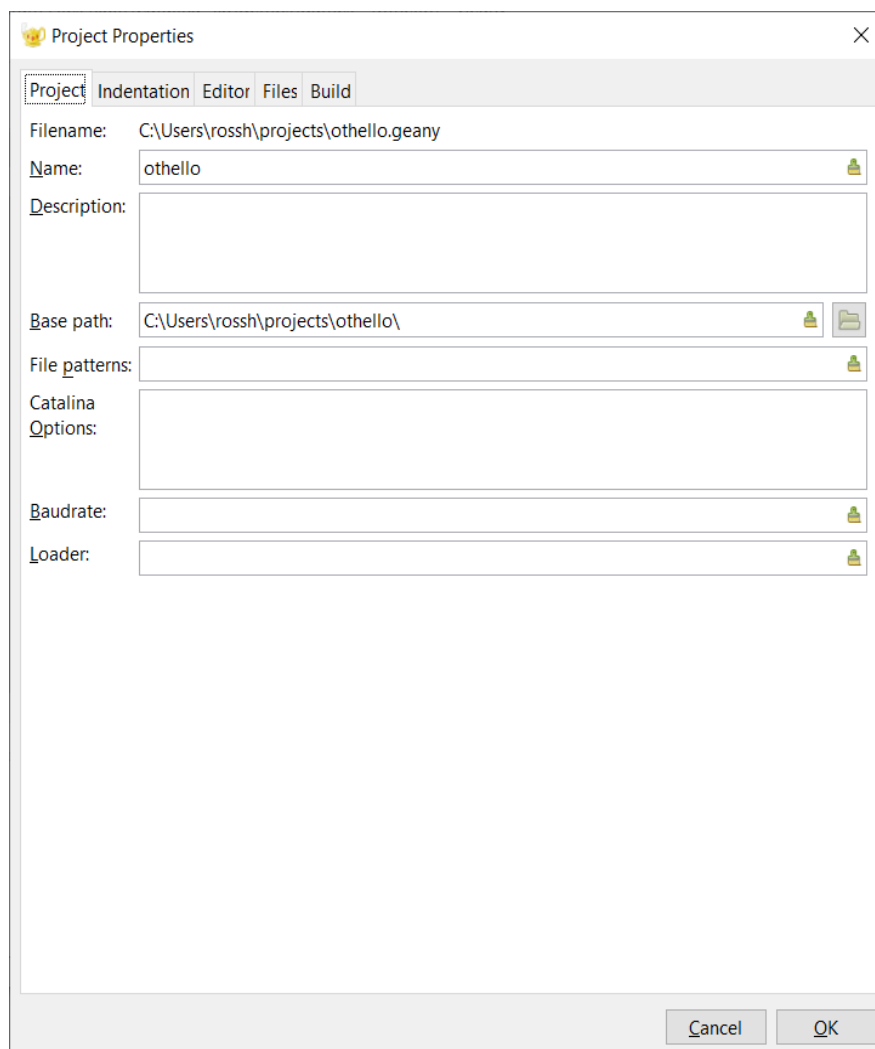
The *othello.c* program will appear in the right hand pane. The symbols defined in the file will appear in the left hand pane (if the Symbols tab is selected in that pane):



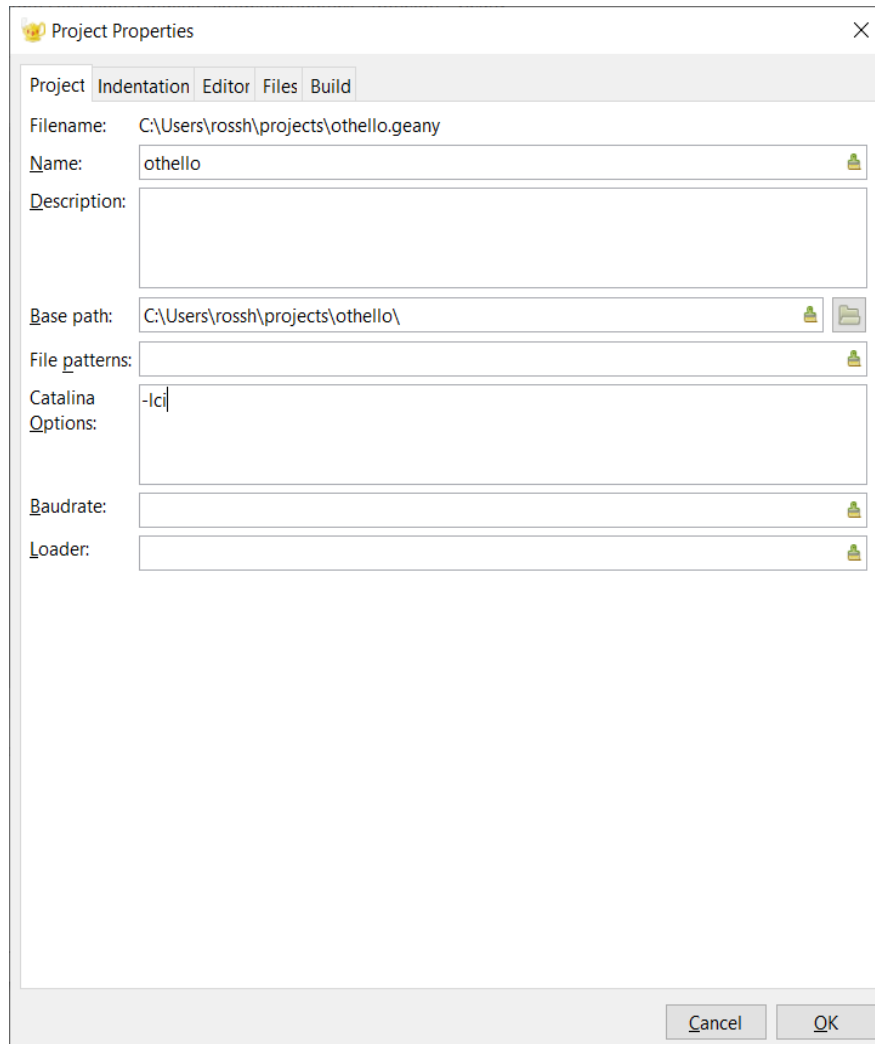
Now there is an **important step** we should do before we go any further. This file is *not yet part of our new project*. To make it so, we need to *save it in the project folder*. Select **File→Save As ...** and navigate to the project folder, then click **Save**.

While you can use Geany to just compile individual files from any folder, it is recommended that when you use projects, you copy all the files in the project to the project folder. If you do not do this, some functions may not work as you would usually expect.

Before we can build the project, we need to set up some Catalina-specific options, such as which C library to use. This is very easy to do – select the **Project**→**Properties** menu item to open the properties dialog box:



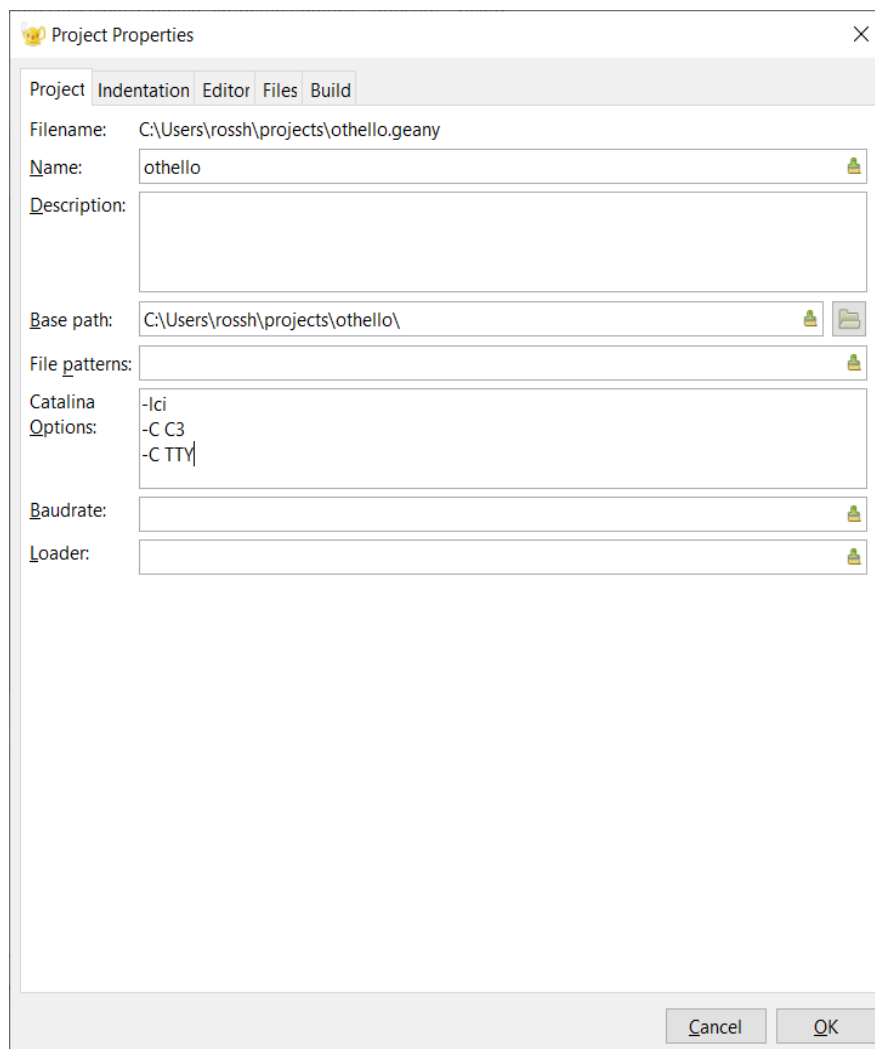
For now, all we really need to do is add **-lci** to the **Catalina Options** field – this tells Geany we want to link our project with the integer version of the standard C library. The dialog box should then look like this:



However, as in the previous example, while we are adding the library option, we can also add any necessary options for our platform and also tell the project we want to use the TTY plugin for serial I/O. This is not required if you are using a standard Propeller platform that is compatible with Catalina's CUSTOM configuration because it is the default, but let's assume you instead have a C3. In that case, you would need to add **-C C3 -C TTY** to the Catalina options.

NOTE: This tutorial assumes you are using a Propeller 1. If you are using a Propeller 2, you should also add **-p2** in the options field, and **230400** in the baud rate field.

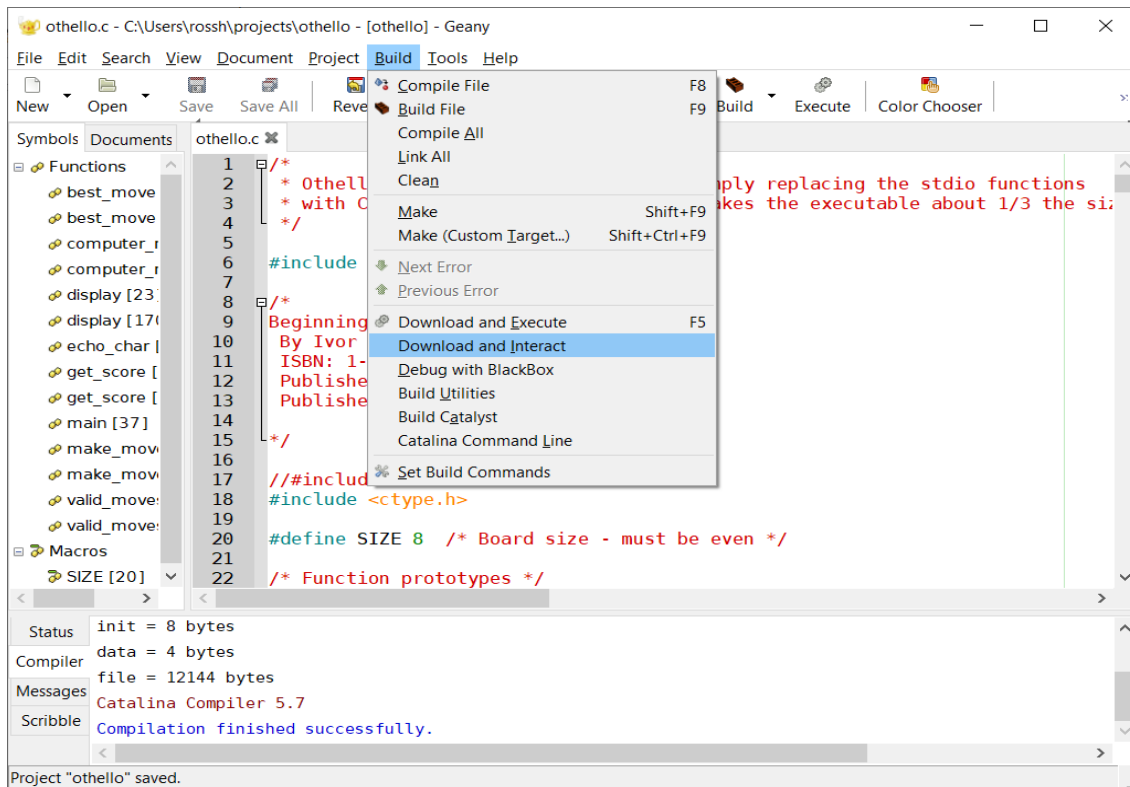
To make the options more readable, you can enter each one on a new line in the Catalina Options field:



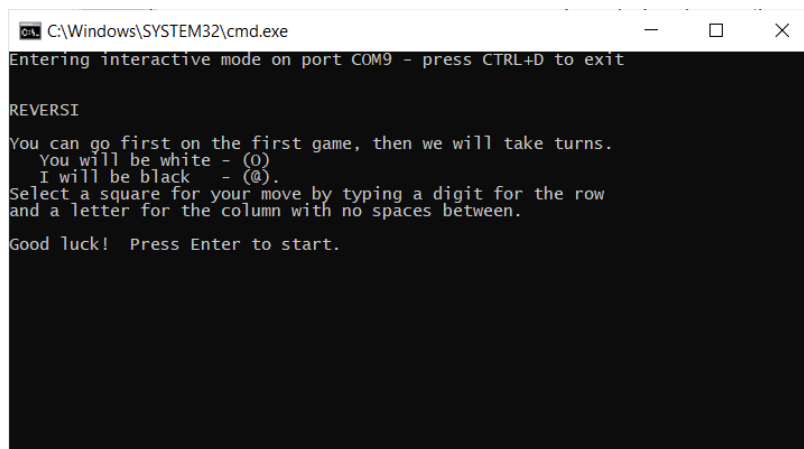
Press **OK** to close the dialog, and then select the **Build**→**Build File** menu item, or press the **Build** button.

The compilation should succeed without any errors.

Now we can run the program. Because we are using serial I/O as our interface, we want to choose the **Build→Download and Interact** menu item. Otherwise, we would not see any output:



When we do so, we should see an interactive terminal window open up, displaying the output from the program:



That's it! A similar procedure can be used to build most of the programs in the "demo" folder. However, you may notice that most of the programs already have Geany projects in their subfolders – for example, in the "demos\games" folder, there is a **startrek.geany** file. If we copied the folder to our own user directory, we could simply open that project file within Geany and

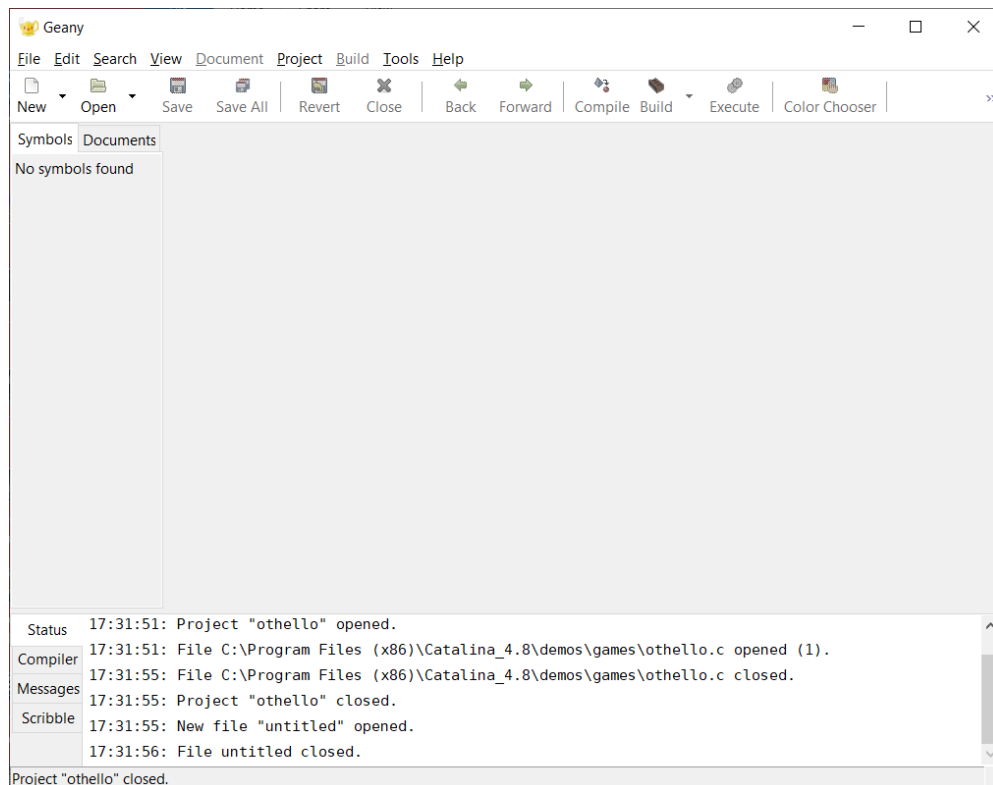
compile the startrek game.

In the next example, we will demonstrate compiling a program that consists of multiple files, and also using the debugger from within Geany.

Using Geany to build and debug a complex program

If Geany is not running, start the **Catalina Geany IDE** from the Start Menu shortcut of that name (on Linux, or from a Catalina Command Line, execute the command **catalina_geany**).

If there is a project open, close it by selecting **Project→Close**, and if there are any C files left open, close them by selecting **File→Close**. You should see a window like this (there may or may not be a file called "untitled" still open – that doesn't matter):

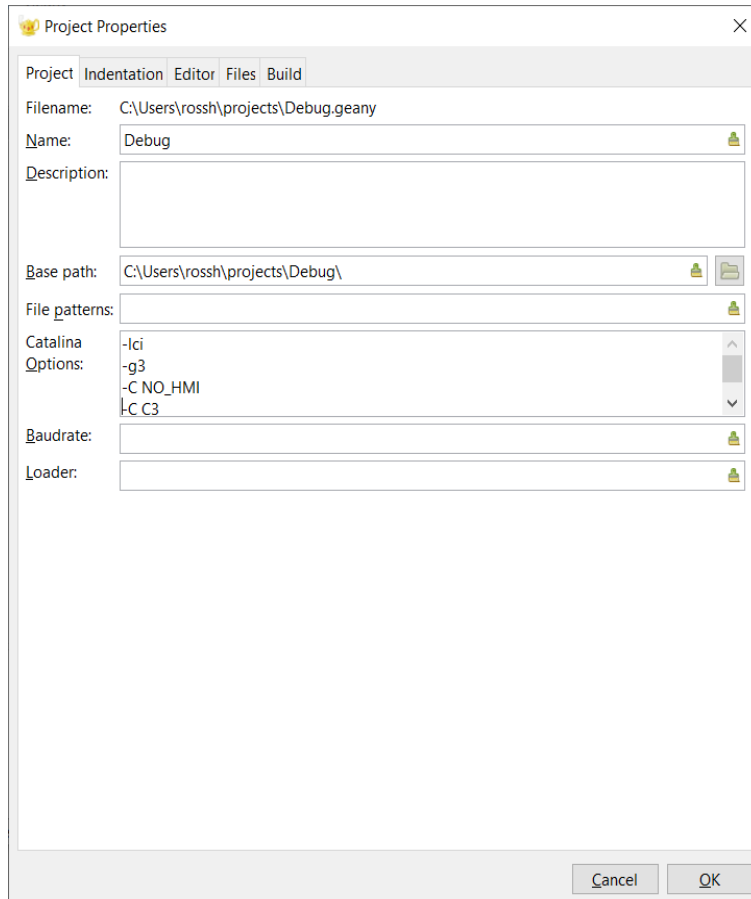


Now we can create a new project using the **Project→New** menu item. This time, we will name our project **Debug**.

The first thing we want to do with this project is tell Geany we want a debug compilation. To do that, select the **Project→Properties** menu item and enter the Catalina Options **-lci -g3 -C NO_HMI**, plus any other platform options (such as **-C C3**) in the **Catalina Options** field.

The **-lci** identifies the C library we want to use, **-g3** enables debugging, and **-C NO_HMI** disables the HMI, which might otherwise use the serial port (which in this case we want to use for debugging). We don't actually care about the output, as this is simply a demo of the compilation and debugging process.

Your project properties should now look something like this:



NOTE: This tutorial assumes you are using a Propeller 1. If you are using a Propeller 2, you should also enter **-p2** in the options field, and **230400** in the baud rate field.

In the last example, we compiled a program that existed in the Catalina demo folder, without copying it into our project. In this example, we are actually going to copy the relevant *.c and *.h files to our project folder. This is what you would normally want to do to modify the files, and in this case it is required to allow us to use the **Compile All** and **Link** commands (which we must use instead of **Build** to compile this project).

The easiest way to copy the files is in a **Catalina Command Line** window. Open a Catalina Command Line using the Start Menu shortcut of that name, then execute the following commands:

On Windows:

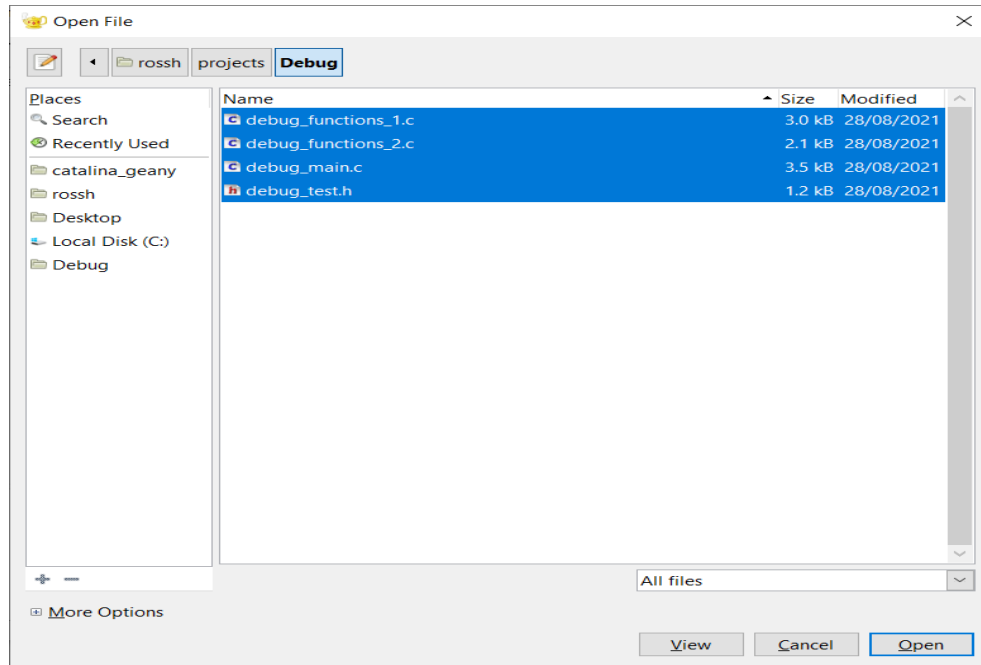
```
copy demos\debug\*.c %HOMEPATH%\projects\Debug
copy demos\debug\*.h %HOMEPATH%\projects\Debug
```

On Linux:

```
cp /opt/catalina/demos/debug/*.c $HOME/projects/Debug
cp /opt/catalina/demos/debug/*.h $HOME/projects/Debug
```

Now in Geany you can select **File→Open**, or press the **Open** button to open the files we just

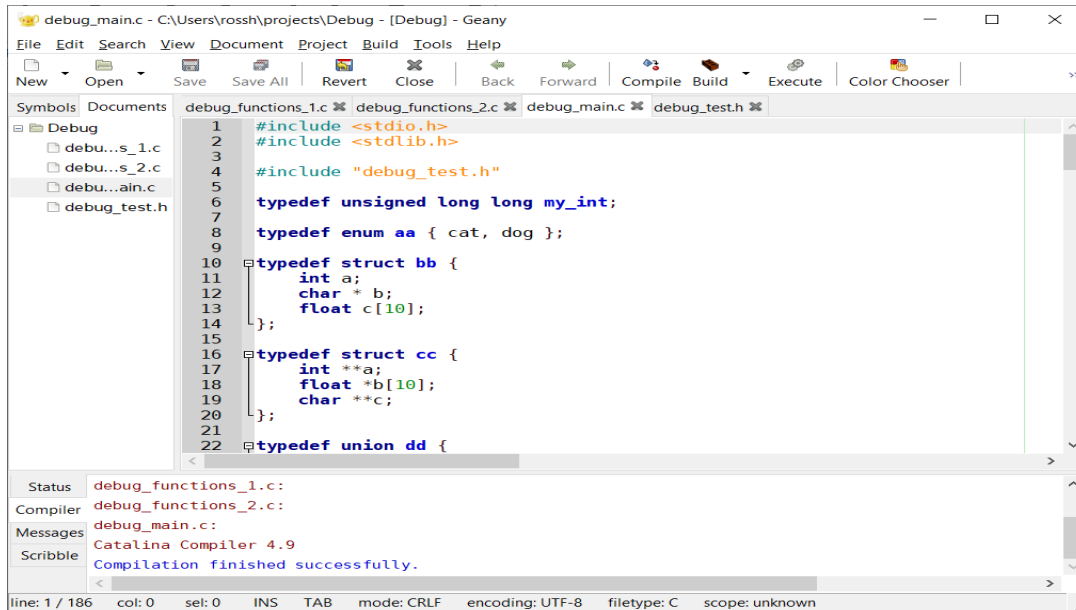
copied to the project folder. Note that you can select multiple files to open at once:



Unlike some other IDEs, Geany has no knowledge about project dependencies, and no built-in knowledge of how to make complex programs that consist of multiple source files. However, we can simply compile all the C files that are in the project folder by selecting the **Build→Compile All** menu item (note that you have to make sure one of the C files is the current file to see this command on the Build menu. For instance, if you have the file “debug_test.h” file selected, you will not see a **Compile All** command). In this case, this is exactly what we want to do. However, to use the **Compile All** command, the program files to be compiled must exist in the project folder - they cannot just be references to files stored elsewhere, as we did in the previous example. This is why we had to copy them into the project folder.

If simply compiling *all* the C source files is *not* all you need to do to build your program, or they have to be compiled or otherwise processed in a particular order, then you will have to use the **Compile** command on each of the C files individually, or use a Makefile (this is described later).

When we use **Compile All** to compile the source files, we may get some warning messages, but the compilation should complete successfully:



Now, we have compiled all the C files in the project (into ".obj" files under Windows, or ".o" files under Linux), but we have not yet linked them together in an executable binary.

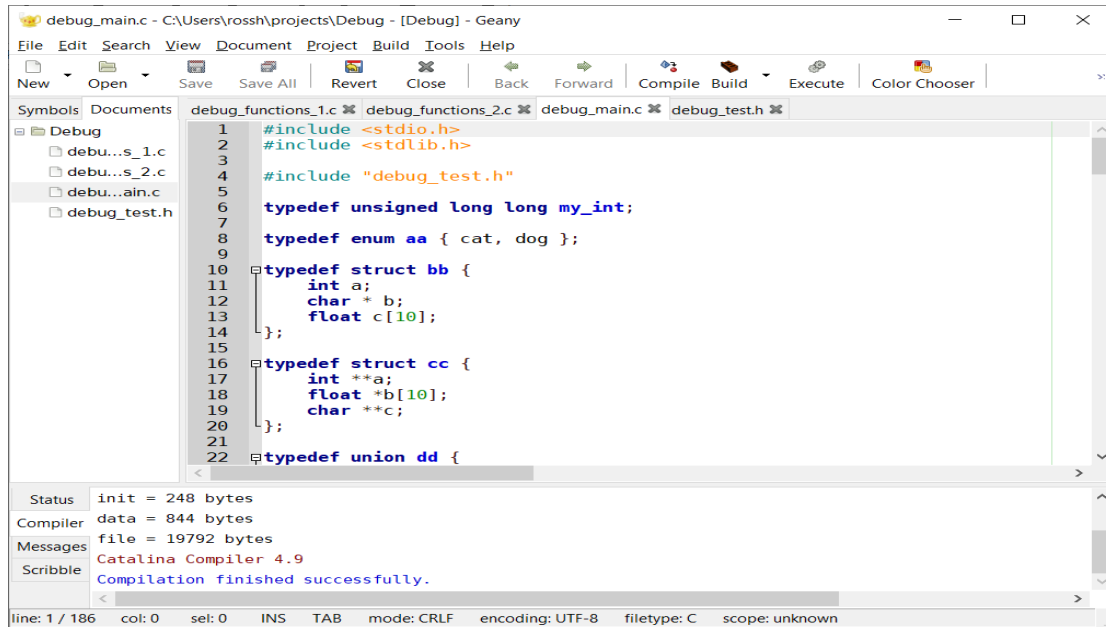
Before we do so, it is important to select the "main" C file, because Geany uses the name of that file for the output binary. So, in this case, select the file *debug_main.c* in the left hand pane, or using the tabs at the top of the right hand pane, and then select **Build**→**Link All**. The project should link all the compiled objects into an executable binary (in this case, it will be called *debug_main.binary*).

The use of the **Compile File** (or **Compile All**) command and then the **Link All** command is an alternative to the single-step **Build** command, and is generally required for complex programs that consist of multiple source files.

Also, note that the **Compile File** command compiles the currently selected open file – which may not be part of the project – whereas the **Compile All** command builds all the files in the project folder.

When you use the **Compile File** or **Compile All** commands, you may sometimes get warning messages – this is because some of the Catalina options (such as libraries) are not actually required during the compilation phase, only during the linking phase – and so you may see messages saying that some options are being ignored. This is normal.

After executing the **Link**, this is what you should see:

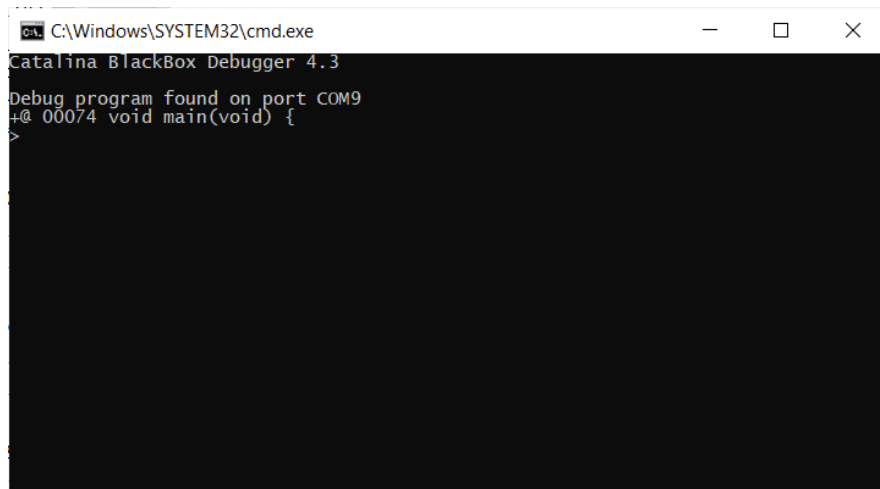


The next step is to download the program for debugging. Since we don't want a terminal to see any I/O, we do that by selecting the **Build→Download and Execute** menu item, or just pressing the **Execute** button. Again, it is important to ensure the main program file is selected in the left hand pane, as this will be used to determine the name of the binary file to be downloaded.

When the download is complete, close the window (you should see the message **(program exited with code: 0)** - this message refers to the *download*, not the program itself).

NOTE: This tutorial assumes you are using a Propeller with a non-serial HMI option (e.g. **TV** or **VGA**). If you are using a Propeller that has only a serial HMI option, you will need a second serial cable connected for Blackbox to use. By default, Catalina is configured to use pins 30 & 31 (on the Propeller 1) or pins 62 and 63 (on the Propeller 2) for the BlackBox port. To use these pins for your serial HMI you will need to modify the pins BlackBox uses – see the **BlackBox Reference Manual** for details.

Finally, select **Build**→**Debug with BlackBox**. You should see a window like this:



We can now interact with the BlackBox debugger. For instance, enter **n <CR>** to step through the program line by line. To step into a function on the line about to be executed, instead enter the command **s i <CR>**. Enter **h <CR>** for help, or **q <CR>** to quit.

There is a separate document ("**Getting Started with BlackBox**") that gives more information about using the BlackBox debugger, and it uses this demo program as an example.

Using Geany to build and run a program using XMM RAM

NOTE 1: To execute this example, you must have a Propeller platform supported by Catalina that has XMM RAM. This example assumes you are using a Credit Card Computer (C3).

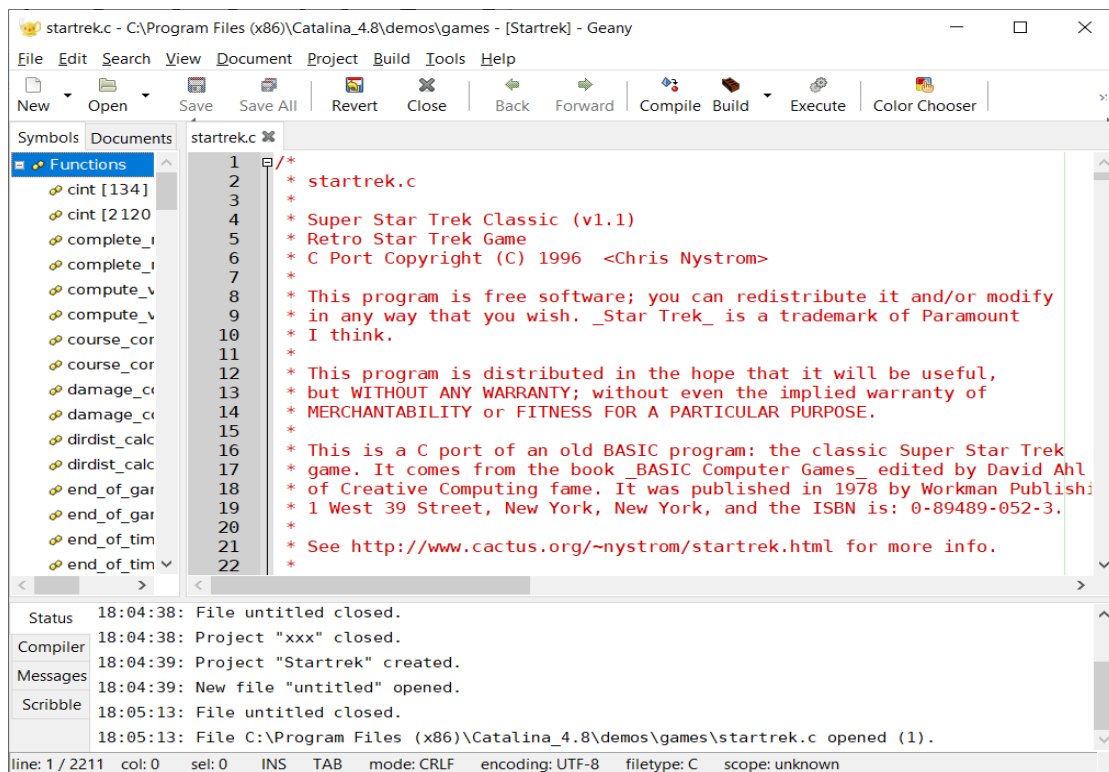
NOTE 2: This example assumes you have worked through the previous examples, so it does not give screenshots of all the screens you will see along the way.

If Geany is not running, start the **Catalina Geany IDE** from the Start Menu shortcut of that name (on Linux, or from a Catalina Command Line, execute the command **catalina_geany**).

If there is a project open, close it by selecting **Project→Close**, and if there are any C files left open, close them by selecting **File→Close**.

Create a new project by selecting **Project→New** and enter the name **Startrek**.

Now select **File→Open**, or press the **Open** button. Navigate to the "demos/games" directory where Catalina is installed (typically, *C:\Program Files (x86)\Catalina\demos\games* on Windows, or */opt/catalina/demos/games* on Linux) and select the file *startrek.c* from this directory, as shown below. Then press **Open**. You should see this:



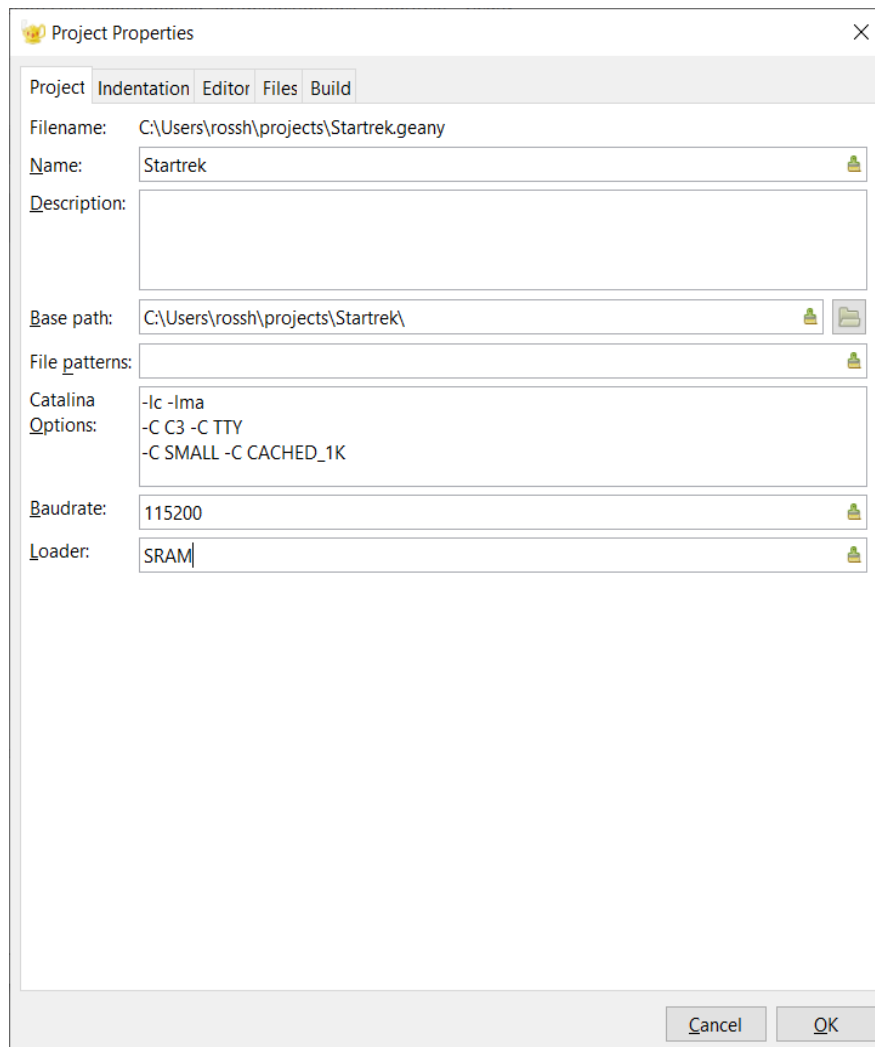
Save this file in your project folder. Select **File→Save As ...** and navigate to the **Startrek** project folder and press **Save**.

Next, we must set the project's Catalina Options. Select **Project**→**Properties**.

For this program, we must use the standard C library (**-lc**) and also a maths library (**-lma**). We must also specify the propeller platform to use (**-C C3**) and that we want to use a serial HMI option (**-C TTY**). Finally, we must also specify the memory model (**-C SMALL**) and the caching option to be used (**-C CACHED_1K**).

For the first time, we also need to specify something in the **Loader** field. We will specify **SRAM** here, because we are compiling the program to execute from SRAM (this platform also has FLASH that we could use).

So your project properties will end up looking something like this:



Now we are ready to build the project. Select **Build**→**Build File** or press the **Build** button. The program should compile successfully.

However, before we can execute the program, we must build the XMM utilities (if we have not already done so). We can do that from within Geany. Select **Build**→**Build Utilities**. A terminal window should appear:

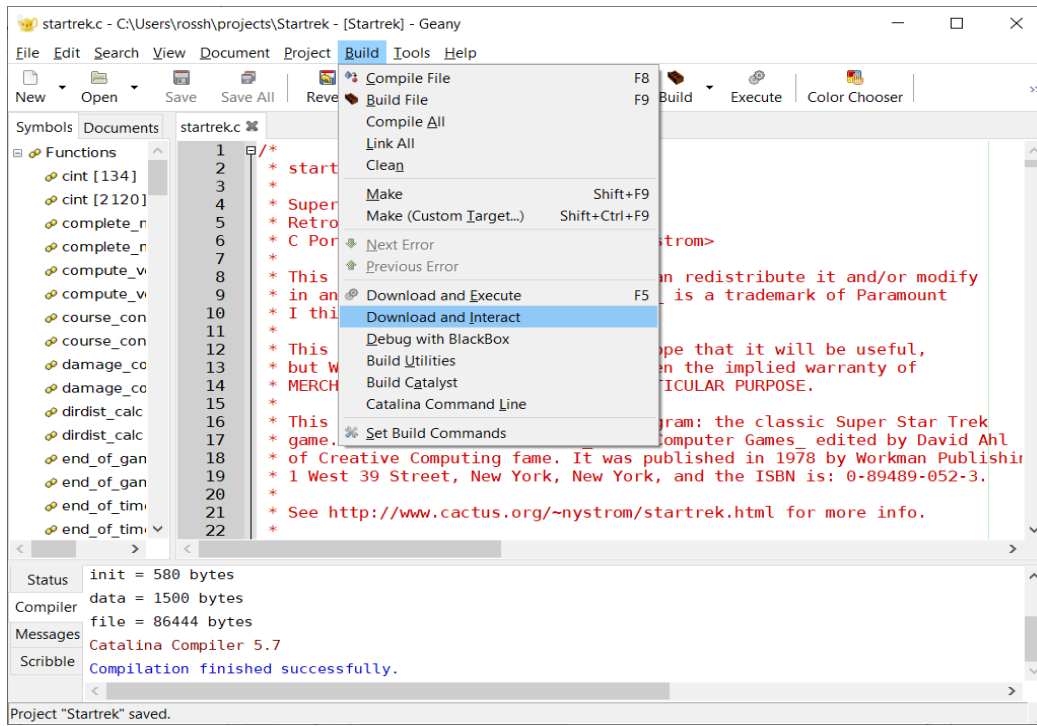
You must step through the build_utilities program, specifying the following options:

- Your platform (**C3**)
- Your XMM board (in the case of the C3 the XMM RAM is built-in, so you do not need to specify anything here)
- Your FLASH cache size (**1**)
- Your FLASH Boot options (nothing is required here)
- Your SRAM cache size (**1**)
- Whether you want to use FLASH or SRAM by default (**S**).

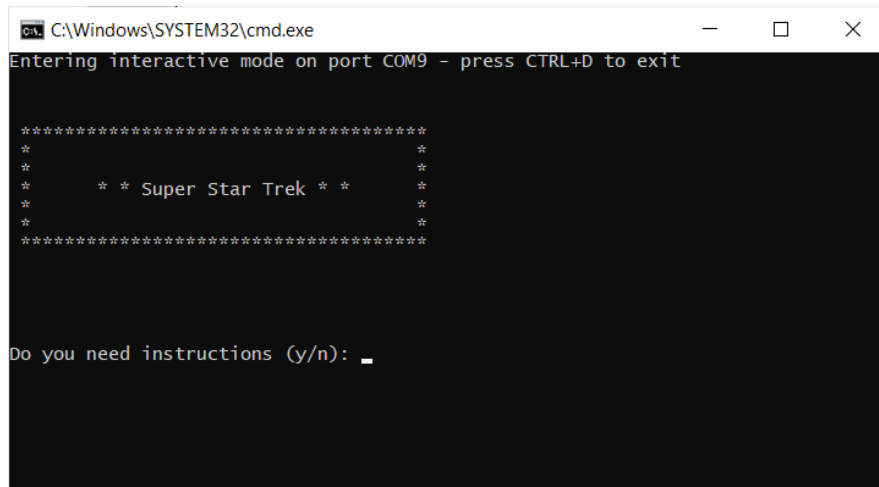
The build_utilities window will close automatically when it is complete.

It is important that the options entered in build_utilities match the options in the Project Properties – in particular, the platform and the cache size.

Now we are ready to download the program – since we want to see the serial I/O, we do so by selecting **Build→Download and Interact**:



You should see the program download using the first and second loaders (the SRAM loader will be used automatically here), and then an interactive terminal window will appear:



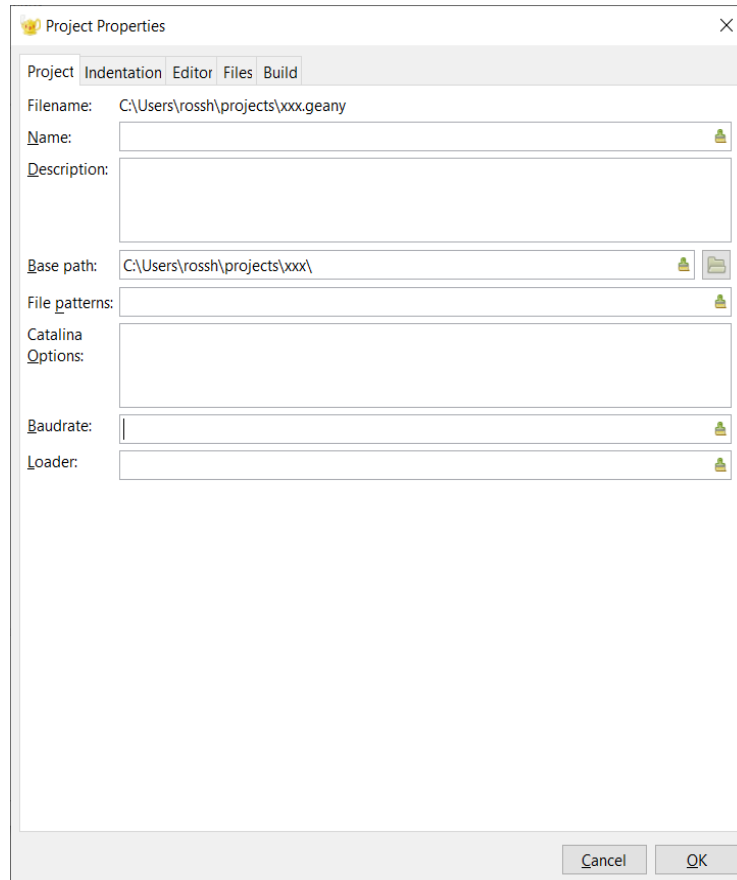
You can build other Catalina demo programs that require XMM RAM using similar steps.

Additions to the Geany IDE for Catalina

Geany is a very flexible IDE. However, some minor changes have been made to better support the Catalina C compiler. These are in the form of Catalina specific additions to the standard Geany project properties, and Catalina specific build commands.

Catalina Project Properties

We have already seen the main changes to Geany for Catalina - they are the new fields on the Project tab of the Project Properties dialog:



The three additional fields on this dialog, and their use, are as follows:

Catalina Options: This is a multi-line text box that is used to specify all the project options. The options should be separated by spaces or new line characters (i.e. not commas). For example:

platform options: **-C C3**, **-C CUSTOM**, **-C P2_EVAL** etc

library options: **-lc**, **-lci**, **-lcx**, **-lcix**, **-lm**, **-lma**, **-lmb**, **-lmc**, **-lthreads**, **-linterrupts** etc

memory model options: **-C TINY**, **-C LARGE**, **-C SMALL**, **-C COMPACT**, **-C NATIVE** etc

miscellaneous options: **-p2**, **-O5**, **-g3**, **-e** etc

See the Catalina documentation for a full set of such options. The value of this field can be used in build commands using **%o** (see below for the Catalina build commands).

Baudrate

This is the baud rate for payload to use. On the Propeller 1, this would usually be **115200** (which is the default), but on the Propeller 2 it may need to be manually specified (e.g. as either **0** or **230400**). The value in this field can be used in build commands using **%b** (see below for Catalina Build Commands). Specifying **0** as the baud rate means to use the default appropriate to the chip (i.e. **115200** for the Propeller 1, or **230400** for the Propeller 2), and this is recommended unless you have a specific need to use a different baud rate.

Loader

This is used for **XMM** and **EEPROM** programs that require a special loader (which must be built using the **build_utilities** batch script). Possible values include **XMM**, **SRAM**, **FLASH**, or **EEPROM**. This field should be left blank for normal Propeller Hub RAM programs. The value in this field can be used in build commands using **%x** (see below for Catalina Build Commands).

Note that the Propeller 2 does not currently require any specially built loaders for loading **FLASH**. Instead, on the Propeller 2 flash is loaded using a special payload script called **flash_payload**. To use it, edit the project Build commands and replace the **payload** command with **flash_payload**. The order of parameters is important here – the first parameter should be the name of the executable to be loaded. So you might use a **flash_payload** command like the following:

```
flash_payload %e -b%b
```

One of the existing Geany fields has been modified slightly for Catalina:

Base path

For files that belong to a project, Catalina Geany has been modified to use file names relative to the **Base path**. This allows Geany project files to be independent of where they are installed. This means, for example, that you can more easily copy all or part of the demos directory to another location to build the example programs. You no longer need to build them in place in the Catalina program folder. While it is not a Catalina modification, note also that in Geany you can specify **./** as your project base path. This applies on both Linux and Windows (even though Windows doesn't normally use **/** as a path separator) - it is not really a path, it is a signal to Geany to use the path to the Geany project file as the base directory for the project. If you intend to use this feature, you should set Geany's **Use project-based session files** and **Store project file inside the project base directory** options. See the section in this document called **Using the Supplied Catalina Geany Project Files** for details on how to do this.

Catalina Build Commands

The other change to the standard Geany IDE is in the form of customized build commands. Here is the Build tab of Catalina's version of the Project Properties dialog for C programs (this is the Windows version – the Linux version has very slight command differences). Note that you have to have a C program open and selected in the main pane to see the C commands:

The screenshot shows the 'Project Properties' dialog box with the 'Build' tab selected. The dialog is divided into three main sections: 'C commands', 'Independent commands', and 'Execute commands'. Each section contains a table with columns for '#', 'Label', 'Command', 'Working directory', and 'Reset'.

#	Label	Command	Working directory	Reset
C commands				
1.	Compile File	catalina %o -c "%d%\%f"	%p	[Reset]
2.	Build File	catalina %o "%d%\%f" -o "%e"	%p	[Reset]
3.	Compile All	catalina -c %o *.c	%p	[Reset]
4.	Link All	catalina %o *.obj -o "%e"	%p	[Reset]
5.	Clean	cmd.exe /c "del /f /q %e *.binary *.eeprom"	%p	[Reset]
6.				[Reset]
Error regular expression:				
Independent commands				
1.	Make	make -C "%p" "%n" CATALINA_OPTIONS=	%p	[Reset]
2.	Make (Custom Target...)	make -C "%p" CATALINA_OPTIONS="%o"	%p	[Reset]
3.				[Reset]
4.				[Reset]
Error regular expression:				
<i>Note: Item 2 opens a dialog and appends the response to the command.</i>				
Execute commands				
1.	Download and Execute	payload %x "%p\%e" -b%b	%p	[Reset]
2.	Download and Interact	payload %x "%p\%e" -i -b%b	%p	[Reset]
3.	Debug with BlackBox	blackbox "%p\%e"	%p	[Reset]
4.	Build Utilities	build_utilities		[Reset]
5.	Build Catalyst	build_catalyst		[Reset]
6.	Catalina Command Line	cmd.exe /k "%LCCDIR%\use_catalina"	%p	[Reset]
%b, %d, %e, %f, %p, %l, %n, %o, %x are substituted in command and directory fields, see manual for details.				

At the bottom right of the dialog are 'Cancel' and 'OK' buttons.

The first thing to note is that Catalina has *more* customizable build commands than the standard Geany IDE. This reflects the many more things that must typically be done to support a separate external processor.

The default values for the customized build commands for C files are as shown. (in grey text – if they are modified from the default commands, they are shown in black text).

Note the use of the standard Geany IDE placeholders: **%d**, **%e**, **%f** and **%p**, plus the Catalina-specific additions: **%b**, **%n**, **%o** and **%x**. These provide access to the Catalina specific Geany project extensions, as follows:

%n The project name. This is intended to be used to select a named target (i.e. in Makefiles that can build more than one target in the same directory).

%b The Baud Rate. Typically **115200** for the Propeller 1, or **230400** for the Propeller 2. It can be specified as **0** to mean payload should use the default appropriate to the type of chip detected.

%o The Catalina Options (as a single string, with newlines removed).

%x The Loader to use (e.g. **SRAM**, **FLASH**, **XMM** etc).

The build commands can be customized per project if required.

Note that Geany is an open source IDE. The source code for the changes to the standard Geany IDE are included in each Catalina release. See the file *catalina_geany_source.zip* in the *catalina_geany* subdirectory.

The Geany Build Menu

Geany has two distinct sets of build commands on the Build menu, and one set of execute commands. Note that in Geany the build menu is determined by the type of file you have currently selected. You must have a **C** file (i.e. a file with a “.c” extension) open and currently selected in the main pane to see the **C** commands.

The three sets of commands are:

The **C** commands

The **Independent** commands

The **Execute** commands

The C commands

The first set of commands on the Build menu are file-based build commands specific to C programs. Note that you must have a C program open and selected in the main pane to see them:

Compile File	compile the current C file only and produce an object file.
Build File	compile the current C file only and link it to produce an executable.
Compile All	compile ALL the C files in the project directory to separate object files, but do not link them. If the files in the directory belong to different projects or must be built with different options then use Make instead (see below).
Link All	link ALL the object file in the project file to produce an executable. The executable will be named for the current file. There must be only one file in the directory with a C main() function. If there is more than one (e.g. if the directory contains more than one project or the project has more than one executable) then use Make instead (see below).
Clean	remove all binary, object, listing and executable files in the project directory (similar to 'make clean').

These commands are suitable for programs that consist of a single C file, or when all the C files in the directory belong to the same project and there are no dependencies between them, and no other steps required except compiling and linking. All the examples described in this document so far have used these commands.

Independent commands

After the simple C commands on the Build menu, you will see the two **make** commands. They are called “independent” because **make** can be used to build *any* type of program, not just C programs:

Make	Make the current project using the Makefile in the current directory – i.e. invoke make specifying the current project as the target.
-------------	--

Make (Custom Target) Make a custom target. Geany will prompt for the target name, which can be the special targets **clean** or **all**.

Using **make** is necessary when there is more than one project in the same folder, or if the project has more than one executable that needs to be built. In such cases you need to write a custom *Makefile* and then use Make to build each project. An example Makefile is given later in this document, and there is an actual example in the Catalina *demos\games* folder.

To use the **make** build commands, you need to have a version of **make** installed. On Linux, use your package manager to install **make** if it is not already installed. On Windows you may have elected to install **make** and a few other utilities when you installed Catalina. If you did not allow Catalina to install them, you can download just the **make** utility from here:

<http://gnuwin32.sourceforge.net/packages/make.htm>

Remember to add the necessary path to **make** to your **PATH** environment variable.

Like all Catalina compilations, Geany will use any Catalina symbols defined in the **CATALINA_DEFINE** environment variable. You can set this variable to specify your platform and memory model. For instance, on Windows you might say:

```
set CATALINA_DEFINE=P2_EVAL COMPACT
```

On Linux, the syntax to set environment variables is different. You might instead say:

```
export CATALINA_DEFINE="P2_EVAL COMPACT"
```

Then, no matter which method you use, your programs will be compiled as if you had specified the options **-C P2_EVAL -C COMPACT** on the command line. Note that if you specify conflicting symbols as parameters to the `build_all` script, it will print a message about the conflict and not compile any programs.

In order to make the options specified in the project properties accessible to the **make** commands, the command actually executed by the Catalina version of Geany is as follows:

```
make "%n" CATALINA_OPTIONS="%o"
```

This allows Geany to pass the options defined for the project to the project Makefile using the variable **CATALINA_OPTIONS**.

Note that if you are intending to use **CATALINA_DEFINE** you should set its value before starting Geany. You can use a combination of **CATALINA_DEFINE** and specific project options to pass information to the **make**.

Execute commands

The remaining commands on the Build menu are utility or execution related commands:

Download and Execute	download the program executable using payload.
Download and Interact	download the program executable using payload and then enter interactive mode.
Debug with Blackbox	download the program executable and then start the blackbox debugger.
Build Utilities	execute the build_utilities script to build the Propeller 1 platform-

specific utilities. This is an interactive script that will ask you for details of your platform. These utilities are not required for the Propeller 2. Note that executing this script requires write permission to the Catalyst installation tree. If you do not have this, you should instead copy the **utilities** folder to your own user directory and compile the utilities there from a command line using the **build_utilities** script.

Build Catalyst

execute the **build_catalyst** script in the to build the Catalyst program loader. This is an interactive script that will ask you for details of your platform. Note that executing this script requires write permission to the Catalyst installation tree. If you do not have this, you should instead copy the **catalyst** folder to your own user directory and compile Catalyst from a command line there using either the **build_catalyst** or **build_all** script. See the *Catalyst Reference Manual* for more details.

Catalina Command Line

open a command line window and set up the command line environment for Catalina.

An example Makefile

The following is an example of a Makefile that could be used either from within Geany, or in a Catalina Command Window. In this case, it is for building the *hello_world.c* program. After the Makefile itself, some of the important things to note are identified:

```
#
# Common Makefile setup - detect P1/P2 and Windows/Linux
# =====

# assume P1
B=.binary

# P2 if there is a P2 in CATALINA_DEFINES
ifeq (P2, $(findstring P2, $(CATALINA_DEFINE)))
    B=.bin
    CFLAGS += -p2
endif

# P2 if there is a P2 in CATALINA_OPTIONS
ifeq (P2, $(findstring P2, $(CATALINA_OPTIONS)))
    B=.bin
    CFLAGS += -p2
endif

# P2 if there is a -p2 in CATALINA_OPTIONS
ifeq (P2, $(findstring -p2, $(CATALINA_OPTIONS)))
    B=.bin
endif

# set up options specific to Windows or Linux
ifeq ($(OS), Windows_NT)
    E=.exe
else
    E=
endif

#set up some common utilities
RM=-rm -f
CP=cp -f

#
# Default Rules and Options
# =====

#set up default options if none specified
ifeq ("$(CATALINA_OPTIONS)", "")
    CFLAGS += -lci -C NO_REBOOT
else
    CFLAGS += $(CATALINA_OPTIONS)
endif

# set up common options
B1=.binary
```

```

B2=.bin
E1=.eeprom
O=.obj
L=.lst
CC=catalina

#set up pattern rules
%.binary : %.c
    -@catalina_env
    $(CC) $(CFLAGS) -o $@ $<

%.eeprom : %.c
    -@catalina_env
    $(CC) -e $(CFLAGS) -o $@ $<

%.bin : %.c
    -@catalina_env
    $(CC) $(CFLAGS) -o $@ $<

#
# Named Targets
# =====

# Named Targets are primarily for use with Geany, and should correspond
# to Geany project names in this directory. Using $B as the suffix in a
# named target means this rule chooses the appropriate binary suffix
# for a P1 or P2 based on whether "P2" exists in CATALINA_DEFINE.
# The first such target will be the default target of this Makefile

.PHONY : hello_world
hello_world : hello_world$B

# add this to the 'all' target
all :: hello_world

# =====

.PHONY : hello_world_1
hello_world_1 : hello_world_1$B

# add this to the 'all' target
all :: hello_world_1

# =====

.PHONY : hello_world_2
hello_world_2 : hello_world_2$B

# add this to the 'all' target
all :: hello_world_2

# =====

```

```
.PHONY : hello_world_3
hello_world_3 : hello_world_3$B

# add this to the 'all' target
all :: hello_world_3

# =====

.PHONY : blink
blink : blink$B

# add this to the 'all' target
all :: blink

#
# Clean Target (should not be first target!)
# =====

# clean target
clean ::
$(RM) *$(B1)
$(RM) *$(B2)
$(RM) *$(E1)
$(RM) *$L
$(RM) *.dbg
$(RM) *.debug
```

Things to note about this Makefile:

1. It attempts to set the executable file extension correctly for a Propeller 1 or Propeller 2, based on whether the string “P2” exists in **CATALINA_DEFINE**, and/or whether “-p2” is specified in **CATALINA_OPTIONS**.
2. It uses default Catalina options in case nothing is passed in **CATALINA_OPTIONS**. This allows the Makefile to be used as easily from the Catalina Command Line (i.e. by invoking **make** manually) as from Geany.
3. There are default pattern rules defined that know how to make **.bin**, **.binary** or **.eeprom** executables (**.bin** for the Propeller 2, **.binary** or **.eeprom** for the Propeller 1).
4. There is one named target – i.e. “hello_world”. So if the Catalina Geany project in which this Makefile is used is called “hello_world”, this is the target that will be built by the **Make** build command in Geany. Other named targets can also be specified in the Makefile, and this allows the same Makefile to be used in other projects. For an example of this, see the Makefile in the “demos\games” folder. Using named targets in this way is why it is recommended *not* to use spaces in Catalina Geany project names – it may be possible to do so, but it complicates naming the targets in the Makefile, and may not work with some versions of **make**.

Using make from the Catalina Command Line

The Makefiles provided for all the demo programs check whether **CATALINA_OPTIONS** have been passed, and try and use sensible default options if not. So even if you are using **make** stand-alone from a Catalina Command Line, it is not usually necessary to specify options on the command line. So you can say (for example) just:

```
make hello_world
```

instead of having to say:

```
make hello_world CATALINA_OPTIONS="-lci -C NO_REBOOT"
```

Like the build_all scripts in the previous releases, all the Makefiles provided in the demos folder attempt to determine automatically if you are compiling for a Propeller 1 or 2, based on either the suffix, or (if no suffix is specified) then on whether the substring "P2" appears in any of the Catalina symbols defined in **CATALINA_DEFINE** or if "-p2" appears in **CATALINA_OPTIONS**. This means you do not generally have to explicitly add the -p2 option unless you are compiling for the default platform (and are therefore not specifying any specific platform).

The Makefiles in the demo folders contain default pattern rules that know how to make **.bin**,

.binary and **.eeprom** executables. So, for example, if *hello_world.c* exists in the folder, you can simply say:

```
make hello_world.bin
make hello_world.binary
make hello_world.eeprom
```

to build the executable with default options (which can also be specified in the Makefile) and get the expected result. You can use the Makefile in the demo directory as a template for your own Makefiles.

Note that **make** uses only the file date/time to determine if the executable needs to be remade, and is not aware of any changes you may make either to **CATALINA_DEFINE** or to the Geany Catalina Options – so in some cases you need to use 'make clean' to remove old executables, or add the -B option to **make** to force it to rebuild the executable. Or you can specify clean as your first target – e.g:

```
make -B hello_world
```

or

```
make clean hello_world
```

Using the Supplied Catalina Geany Project Files

Many of the C files in the demo directory come with Geany project files which you can just pen within Geany. However, to use them you should set both Geany's **Use project-based session files** and **Store project file inside the project base directory** options.

These options are in the Projects section on the Miscellaneous tab of the Geany **Edit→Preferences** menu:

