AmigaOS 4.1 SDK

AmigaOS 4.1 Final Edition
The AmigaOS SDK Team
http://wiki.amigaos.net/

Table of contents

1. Introduction

Welcome to the AmigaOS 4.1 SDK. This guide should serve as a quick introduction to what you can find here. Note that depending on the specific version of the SDK, some of the options mentioned may not be available.

Installation

Prior to installation, you should make a backup of all files that you modified and/or added to the SDK:Local drawer and delete the old SDK installation. Alternatively install the SDK onto a different disk or in a different drawer. The layout of the SDK has changed and to avoid confusion, this method is preferred.

After the installation

In order to function properly, some paths and assigns need to be setup. This is done in a script file started via the S:user-startup file. It is recommended that you reboot your machine after the installation to prevent any problems with leftovers from a previous SDK installation.

2. What's in the SDK?

After the installation, you will find the following directories rooted from the SDK assign:

- **Documentation.** This drawer contains various documentation, AutoDocs, and other information related to the SDK. It also contains documentation on the various tools and compilers.
- **Examples.** This drawer contains example source code on how to program for AmigaOS 4.x. Examples are sorted by theme/topic.
- **Local.** The Local drawer is our means of isolating the compilers and third-party additions. See chapter 7. The Local drawer for more information on the Local drawer.
- **gcc.** The gcc drawer contains the GNU C/C++ compiler. It is set up in a way that there are no user-serviceable parts inside. This makes it easy to exchange the compiler when later versions become available.
- **vbcc.** Inside vbcc drawer you will find the vbcc compiler, which is a highly optimizing portable and retargetable ISO C compiler. It supports ISO C according to ISO/IEC 9899:1989 and most of ISO/IEC 9899:1999 (C99).
- **Include.** This drawer contains system-level include files. Like the gcc drawer, it should not need to be modified.
- **Tools.** The Tools drawer contains various tools that are useful for conversion of files, for reading the autodocs or for setting the default gcc compiler.

3. Where is all the documentation?

The SDK contains enough documentation to get you compiling AmigaOS programs but it lacks in OS design details, full tutorials, etc.

Full documentation is available on the AmigaOS wiki at http://wiki.amigaos.net and it is expanding all the time. From the wiki you can even produce your own personalized books if you wish fully replacing the old and outdated ROM Kernel Manuals for previous versions of AmigaOS.

4. GCC and the SDK

Since this new SDK v54, we left behind GCC 4 and moved forward to newer and greatest versions of this open-source compiler for our system. From this version, we introduced the ability to install multiple versions of GCC working in parallel. Using the SDK installer you can choose which versions you want to install among GCC 6.4.0, 8.4.0. 10.3.0 and 11.2.0, considering the last two as experimental.

As the default compiler, we recommend using GCC 8.4.0, which is well tested, but versions 10.3.0 and 11.2.0 will produce more optimised and faster code. GCC 6.4.0 is recommended for compiling optimised code for SPE based machines like the A1222 since this is the last version of GCC that supports these CPUs.

In case you select to do a full SDK installation, all four GCC versions will be installed with 8.4.0 being the default one. By default we mean the compiler that will be used when running `gcc` or `c++` or `ppc-amigaos-g++` etc. The order the installer sets the default compiler is 8.4.0, 10.3.0, 11.2.0 and 6.4.0. If you select to not install version 8.4.0, then the next one will be the default one, which is 10.3.0.

In any way, there is a tool available under Tools folder which can help you change the default GCC compiler on the fly without the need for any reboot. This is called `set_defGCC` and you can use it from Workbench or Shell. When you start it from Workbench a window will appear asking you to choose which version of GCC you want to be the default one. As soon as you select one, this is going to be enabled as default and will remain like that even after a reboot of your system. To use it from Shell you just need to execute it with the major version as an argument, i.e. `set_defGCC 11` to set GCC 11.2.0 as the default one. At the end, a confirmation window will show up, mentioning the GCC version that is set as default.

If you want to use the other compiler versions without changing the default one, you can use the required major version beside the command you want to use, i.e. `gcc-11`, `g++-6`, `gcc-ar-8`, `c++-10`, `ppc-amigaos-gcc-10` etc. To compile optimized code for **SPE CPUs** you can use `gcc-6` or `gcc-spe`, as both points to the same version of GCC.

Verify the installation

You can verify that gcc works by entering the command line

```
gcc --version
```

This should show you the version number of the default GCC compiler.

```
gcc (adtools build 8.4.0) 8.4.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

This means that the gcc compiler is available.

To test the compiler, enter the following in a text editor and save it under the name `hello_world.c` in some working drawer:

```
#include <stdio.h>
int main()
{
    printf("Hello, World\n");
    return 0;
}
```

In the shell, change to the directory where you saved the file. Compile it with:

```
gcc -o hello_world hello_world.c
```

After a few seconds the compiler should stop and the shell prompt returns. You should now verify that the compiler produced a binary (`dir` should list a file `hello_world`) in the current drawer. Test it by entering hello_world on the command prompt. The output should say `Hello, World` on a line by itself followed by he shell prompt. Congratulations, you just compiled your first AmigaOS program!

5. C Standard Libraries (*newlib* & *clib2*)

At the heart of every C compiler there is a so-called “C runtime library” which provides the basic functionality required for starting a program in a more or less comfortable environment. The C runtime library contains functions like printf used in the example above.

The SDK comes with two different runtime libraries named `newlib` and `clib2`. By default, the compiler uses the `newlib` C runtime library which is the preferred choice for AmigaOS targeted programs. The `newlib` library can only be used as a shared library and supports the new shared objects. By contrast, the `clib2` library is a statically linked library and is not available as a shared library and does not support shared objects. The `clib2` library also tends to cater more to porting software from other platforms and can be used in debug mode to help find obscure issues. By default, it is recommended to choose `newlib` first which is why it is the default.

The C library can be selected at compile and link time via the `-mcrt=<library>` option. There are currently three possible values for `<library>`

- `newlib`: This uses the `newlib` shared runtime library (default).
- `clib2`: This uses the `clib2` static runtime library.
- `clib2-ts`: This uses the thread-safe variant of the clib2 static linker library. In addition, two other options are available: ixemul and libnix. Both of these options are not currently supported by the SDK.

6. Shared Objects

Shared objects are a relatively new feature added to AmigaOS which greatly simplifies the process of porting software from other platforms and provides developers with an alternative to the traditional AmigaOS shared library.

A shared object library is linked at runtime rather than compile time, so that the code can be exchanged without recompiling the program. Shared objects are very similar to AmigaOS shared libraries, however, there are a few notable differences:

- AmigaOS shared libraries share one data segment (the library base) among multiple openers. Shared objects each have an individual data section per opener/context.
- On AmigaOS, shared object files aren't really shared yet meaning that the code (and the data but for the data this is intentional) is loaded into memory every time the shared object is used.
- Shared objects can export both code and data symbols which can be used by the program using the library. Likewise, shared objects can use code and data symbols in the main program as well as any other library opened by the program.

Typically, shared objects are for:

- Software ports from UNIX platforms. For example, most libraries under Linux aren't straightforwardly converted into an AmigaOS shared library because of the requirement for a private data segment or export of data symbols or shared usage of main program symbols.
- AmigaOS native software that requires named binding or functionality similar to the dlopen/dclose/dlsym mechanism. This can be handled using elf.library directly as well but shared objects make it easier on the developer.

Compiling the shared object library

All the code that goes into a shared object (never the main program) needs the `-fPIC` compiler switch.

A typical command line would look something like:

```
gcc -fPIC -o foo.o foo.c
```

The -fPIC option is extremely important and produces position independent code. This option may be used for both shared objects (.so) and statically linked libraries (.a). This option must never be used on the main program so be careful to apply it only when required as the compiler will not warn you of the mistake. Also note that case is significant and a -fpic option exists which is not applicable here. Please refer to the GCC documentation for more information.

Instead of ar and ranlib which are used for static link libraries, the final shared object library is built by ld. To invoke ld, use a command line like:

```
gcc -shared -o libfoo.so foo1.o foo2.o foo3.o
```

The -shared option instructs the compiler that it is producing a shared object file. This will build shared object libfoo.so from foo1.o, foo2.o and foo3.o.

Compiling the Main Program

The main program is compiled as always and should never use the -fPIC option. However, when linking, you need to add the -use-dynld option and specify the libraries as normal:

```
gcc -o testfoo.o -c testfoo.c
gcc -use-dynld -o testfoo testfoo.o -lfoo
```

The -use-dynld option tells the linker to utilize shared objects when available. If a shared object library with the name libfoo.so is not found then the linker will automatically use the file libfoo.a if found. This behaviour may surprise some developers so be careful to verify that your executable actually is linked to what you believe it is linked to. Read on for more information.

Shared object plugins

Some projects may use shared objects as plugins. This requires the main program to be linked with the option -Wl, --export-dynamic so that the linker can resolve the reverse dependencies with the shared object(s).

Mixing .so and .a Files

It can be somewhat dangerous to mix shared object (.so) and static link library (.a) files for use in the same executable. The problem is that many of the static link libraries available are not exclusively compiled with the -fPIC option which means they do not contain position independent code. Shared object libraries exclusively contain position independent code and expect to call into position independent code. The end result can be an application which crashes whenever a function is called from a `.so` file into a `.a` file. If possible, try not to mix the two unless it is known for a fact that the `.a` files have been compiled with the `-fPIC` option.

It is important to note that the compiler will prefer shared objects but will also default to using static link libraries if a shared object library is not available. This is the normal behaviour when using the `-l` linker option. To avoid it you explicitly specify the link library by its full name (e.g. `libdl.so` instead of `-ldl`). To find out exactly what your application has been linked against it is best practice to produce a linker map using the `-Wl`, `-Map myapp.map` option on the compiler command line and study it.

Runtime Search Path

Unlike the traditional AmigaOS shared library which usually resides in **LIBS:**, a shared object library resides at the **SOBJS:** assignment. The search path executables use for shared objects is as follows:

```
PROGDIR:
PROGDIR:SObjs
SObjs
SOBJS:
```

Shared object files are searched for in the order given above from top to bottom. Keep this in mind if your application is behaving abnormally as a user may have added or moved a shared object file.

Version Control

Version control of shared objects is more complicated than the traditional shared library. Unlike a shared library, a shared object does not contain version information which is checked at runtime. Instead, shared objects rely on the file name for version information at load time. Multiple versions of a shared object will be stored in separate files with different file names. The version which an application chooses is controlled by the linker command line. For example, if the `-ldl` option is used then the `libdl.so` file will be loaded whatever version it may be. In contrast, if `libdl-1.2.so` is specified then the application requires the `libdl-1.2.so` file to be present to run. This gives the developer fine grained control over which version of a shared object is required if necessary. File links are normally used to specify which shared object file is the current version:

```
MakeLink libdl.so libdl-1.2.so SOFT
```

There are also file links from the **SDK:newlib** and **SDK:Local** directory trees into the **SOBJS:** directory for various shared object libraries. The GCC compiler will search for shared objects in the same way it searches for static link libraries (i.e. `.a` files) so links are provided where appropriate.

Distributing Shared Objects

If you distribute a program compiled using shared objects, you may need to include some of them along with your application. Verify that all the shared object files required are present in the default AmigaOS installation you are targeting. Otherwise, the user will experience a failure at runtime when trying to start your application. Always test against your target system to avoid disappointment.

To check what shared object files your application binary requires you may also wish to use the `readelf` command as follows:

```
readelf -d C:Python
```

This will produce output containing lines similar to the following:

```
0x00000001 (NEEDED)        Shared library: [libpython25.so]
0x00000001 (NEEDED)        Shared library: [libexpat.so]
0x00000001 (NEEDED)        Shared library: [libz.so]
0x00000001 (NEEDED)        Shared library: [libdl.so]
0x00000001 (NEEDED)        Shared library: [libpthread.so]
0x00000001 (NEEDED)        Shared library: [libgcc.so]
0x00000001 (NEEDED)        Shared library: [libc.so]
```

Each line containing (`NEEDED`) indicates your application requires that shared object library to run.

V1 versus V2 Shared Objects

Shared objects were not fully functional until the AmigaOS 4.1 Update 1 release which fixed many important issues with the earlier releases. It is recommended that developers target AmigaOS 4.1 Update 1 or higher when using shared objects to avoid complications. Shared objects may be used in previous releases but they can behave erratically and users may become frustrated and blame your application. It is also critical that you use SDK 53.15 or higher which produces shared objects in the new V2 format necessary for correct operation.

To discover if a shared object library is a V1 or V2 file use the readelf command:

```
readelf -d SOBJS:libexpat.so
```

Within the output you will see a line similar to the following:

```
0x6000000e (AMIGAOS_DYNVERSION)         0x2
```

V1 shared object files will lack `AMIGAOS_DYNVERSION` entirely while V2 files will contain the label along with the version number. In this case it is `0x2` which is a V2 shared object file. It is possible to mix V1 and V2 shared object files but the V1 shared objects contain the defect which prevents them from working correctly when a memory jump is too large. Given that AmigaOS scatter loads binaries it is possible your application works and then doesn't work when V1 shared objects are in use.

An additional feature of shared objects is that runtime binding can be deferred until needed which can decrease load times. This feature only works when V2 shared objects are in use.

Not Really Shared

As mentioned in the introduction, shared objects do no currently share code segments between applications which means each instance of an executable will use a copy of the shared code. This is planned to be addressed in a future release of AmigaOS so do not depend on this behaviour. Always assume the code is truly shared.

7. The Local drawer

The goal of the local drawer is to make local additions of include files and libraries possible without having to modify the `gcc` and `include` drawers. That makes it possible to simply replace one of these drawers when a new update becomes available without loosing any locally installed additional packages. The local drawer contains three sub-drawers, one for common files and one for each of the supported runtime libraries.

These drawers are:

- **common.** The common drawer contains files that can be used under both `newlib` and `clib2`. For example, header files for a normal AmigaOS library can be installed in common, since they don't usually depend on the C library being used.
- **newlib.** The `newlib` drawer contains files that can be used with `newlib` only. Note that shared objects can only be used with `newlib` so you will find links to shared objects from this directory tree.
- **clib2.** The `clib2` drawer contains files that can be used with `clib2` only. Usually, these are the headers and library files of static link libraries. Static link libraries usually depend on the C library used and are not interchangeable.

All of these drawers usually contain an `include` and `lib` sub-drawer (with the exception of common, which only contains the `include` sub-drawer). This is the place where the include files and library files are installed for the specific C library.

If you compile for `newlib` (the default) then the paths `SDK:local/common/include` and `SDK:local/newlib/include` are added to the search path for header files. Likewise, the

`SDK:local/newlib/lib` path is added to the search path for shared objects and static linker libraries when linking.

If you compile for `clib2` then the paths `SDK:local/common/include` and `SDK:local/clib2/include` are added to the search path for header files. Likewise, the `SDK:local/clib2/lib` path is added to the search path for static linker libraries when linking.

It is highly recommended that you do not install any additional headers or libraries in other places in the SDK. The next update of the SDK will delete these together with the gcc compiler when installing a new version. Also, if you want to distribute runtime-specific (or common) files (like include files for a shared library you wrote or link libraries ported from UNIX) you should put them in `SDK:Local`.

Besides the drawers mentioned above, `SDK:Local` also contains drawers that are unrelated to the C library in use:

- **C.** Any local commands and binaries should go in here.
- **Documentation.** Any local documentation that comes with third-party add-ons (see below).
- **Source.** This drawer contains source code and diffs for third-party add-ons.

Example packaging for an AmigaOS shared library developer files

For the following example, we suppose that you wrote a library called `cool.library` and want to distribute developer files for it (the packing and presentation of the user files and possible user documentation is outside the scope of this document). Suppose you have the standard set of include files, most notably

```
include/proto/cool.h
include/interfaces/cool.h
include/inline4/cool.h
include/libraries/cool.h
interfaces/cool.xml
Documentation/cool.doc
Documentation/cool_programming.pdf
```

You should package these files in a way that they only need to be unpacked in the root of the SDK (the **SDK:** assign) to be useful for the fellow programmer. In this case, this would look somewhat like this:

```
Local/common/include/proto/cool.h
Local/common/include/interfaces/cool.h
Local/common/inline4/cool.h
Local/common/libraries/cool.h
Local/common/interfaces/cool.xml
Local/Documentation/AutoDoc/cool.doc
Local/Documentation/cool/cool_programming.pdf
```

If you package your files this way, they will be easily accessible as soon as a programmer unpacks the archive in the SDK root (alternatively, an installer script can take care of this). Furthermore, any change in the system headers or gcc compiler drawer will not hamper a programmer's ability to use `cool.library`.

8. The adtools Project

The GCC compiler, binutils and gdb are now part of the adtools project. The project is hosted on GitHub, that tries to unify the development of the developer tools for AmigaOS and some Amiga-like platforms under a common project. The adtools project's home page can be found at https://github.com/adtools. To get the source code of the GPL parts of this SDK, please refer to this web page.

9. Compile SPE compatible code

If you have already read section 4. GCC and the SDK you probably have seen that SPE (Stream Processing Engine) based CPUs are mentioned, and that's the reason why GNU GCC 6 compiler is included in this SDK release. This version is the last known one that supports CPUs, like the P1022, that are not fully compatible with PowerPC FPU command set.

If your program relies on floating-point calculations, it is recommended those parts be compiled with SPE support.

To do that, you have to install the GCC 6 package during the SDK installation and use `gcc-6` or `gcc-spe` when you compile your code. The following arguments are necessary for the process:

```
-mspe -mcpu=8540 -mfloat-gprs=double -mabi=spe
```

After that, a fully SPE compatible binary will be available for you to use.

10. vbcc and the SDK

Since the new SDK v54 release, we brought back the vbcc compiler as part of the development package. vbcc is a highly optimizing portable and retargetable ISO C compiler. It supports ISO C according to ISO/IEC 9899:1989 and most of ISO/IEC 9899:1999 (C99).

vbcc is developed by Dr Volker Barthelmann and the AmigaOS versions are maintained by Frank Wille.

The compiler is ready and configured to be used as soon as you install it through the SDK installer. It can work in parallel with the GNU compiler installation without your need to make any changes in your code.

> **Note**
>
> vbcc license does not allow the release of commercial applications for PowerPC CPUs without the prior written consent from vbcc author Dr Volker Barthelmann. Please, consult chapter 1.2 in the vbcc documentation, included in the SDK under **Documentation/vbcc** folder.

Verify the installation

You can verify that vbcc works by entering the command line

```
vbccppc -v
```

This should show you the version number of the vbcc compiler.

```
vbcc V0.9h (c) in 1995-2022 by Volker Barthelmann
vbcc code-generator for PPC V0.7 (c) in 1997-2022 by Volker Barthelmann
```

This means that the vbcc compiler is available.

vbcc and C standard libraries

vbcc is coming with its own C standard library, called vclib, which is the default one that is used when you compile your source code.

To compile the hello_world.c file you created previously in 4. GCC and the SDK open a shell window, change to the directory where you saved the file and compile it with:

```
vc -o hello_world hello_world.c
```

If you want to use the newlib C library, you need to change the above line to:

```
vc +newlib -o hello_world hello_world.c
```

In both situations, an AmigaOS 4 PPC native binary will be created.

11. Version Control System

Version control, also known as source control, is the practice of tracking and managing changes to software code. Version control systems are software tools that help software teams manage changes to source code over time. As development environments have accelerated, version control systems help software teams work faster and smarter.

Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

In this SDK package, two of the most widely accepted and used version control systems are supported, which are Subversion and Git. Under the C folder, you will find the ported binaries for `svn` that can be used with Subversion servers, and SimpleGit (`sgit`) which is a simple git client based on libgit2. Both are configured to be used out of the box, but there might be some more configuration needed from you. You will find the tools' documentation under the folder **Documentation/Tools**, as well as books about these systems in PDF format, available under the Creative Commons license.

12. Profilers

During the development of an application, you might introduce situations where the code does not perform as well as possible or it needs some optimization. But how will you be able to tell which parts need to be optimised? How can you measure the time spent in a function and which function calls other functions? And how can you measure if the changes you did actually optimised the code or not?

In this SDK we include two performance profiling applications, which are Profyler and hieronymus, kindly contributed by their developers. Profiling applications are useful because they answer all the above questions.

You will find Profyler under the `Tools` folder and hieronymus under `local/C`, as it is used only from the shell. Both applications are native and exclusive for AmigaOS 4, coming with the necessary documentation, which we recommend reading before start using them.

13. Licenses and Copyrights

Some of the programs contained in this SDK are governed by different licenses. Most notably, the GNU compiler tools, the GNU debugger, GNU make and other programs are governed by the GNU General Public License. Some contributions are under different licenses; see the accompanying documentation for details.

You can find a copy of the GNU General Public License in the file COPYING.GNU in the Documentation drawer.

The newlib C library is governed by the newlib licenses. You can view the newlib license in COPYING.NEWLIB in the Documentation drawer.