The Haujobb Amiga Framework

Abstract

If you want to code Amiga AGA demos, are able to program, but don't know how to get started on the Amiga, then this document is for you. Also if you are already more experienced on Amiga and want to learn about how we do our demos. This document describes how to set up and use the Haujobb Amiga Framework on Amiga and PC. The framework allows for prototyping, debugging, and timing effects natively on the PC (currently Windows mainly, but Mac and Linux should work as well); and then cross-compiling for the Amiga. We have used these exact codes and workflows for our demo *Beam Riders*; and prior versions for *Last Train to Danzig* and *Prototype 1*.

This document guides you through setting up the required components in small steps with immediate incentives – leading up to a complete example demo with timing done via the Rocket editor. To get started, first read the guide to the document section to find your way around. Then go and make a demo about it!

Table of Contents

Part I Setup

Part I Setup

The Haujobb Amiga Framework facilitates modern Amiga demo making on several levels.

On its lowest level, WickedOS, it provides with you with an API for assembly-level access to over 20 different chunky to planar based screenmodes, several music replayers, and interrupt hooks. You can choose from using a multi-tasking-friendly hardware abstraction layer, or a traditional multi-tasking-killer mode. WickedOS has been used in a number of our Amiga demos, ranging from *Mnemonics* to *Beam Riders*. By default it targets “todays” 68060 AGA demos, i.e. having an FPU and at least a 68020 or better CPU in the target machine.

On a higher level, the framework allows you to mix assembler and C-code. Furthermore, it provides you with a reimplementation of the WickedOS-API in Qt, This allows for developing and debugging effects natively on the PC, before cross-compiling (and further optimizing) them for the Amiga.

1 Guide to the document

This document tries to document to entire framework, which includes working on the PC and well as on the Amiga (with different tools).

If you just wanted to have a quick start on your PC, read the Quickstart section elsewhere.

If you also wanted to set up assembler, C-compiler and linker on the Amiga, start reading at that chapter elsewhere. Please note that albeit those tools work perfectly fine on Amiga, we are not supporting them fully with our makefiles.

If you were really curious and also wanted to play around with WickedOS using Asm-One and Devpac, you could read the corresponding section elsewhere in the annex.

In any case, once you installed the assembler, compiler, linker and make tools, you should first check out the Hello World example elsewhere, before following the rest of the examples, which are described in Part II of this document, starting elsewhere.

1.1 Resources

The complete sources are available at https://github.com/leifo/haujobb-amiga. The documentation is hosted at http://www.dig-id.de/amiga/framework/, PDF-version at http://www.dig-id.de/amiga/framework/haf.pdf.

1.2 Quickstart (TL;DR)

The Haujobb Amiga Framework supports development of Amiga demos on PC, and then cross-compiling them for the Amiga.You probably didn't want to compile and assemble on the Amiga anyway, so you just:

- set up the vbcc cross-compiler for Windows using Install-VBCC.exe (to c:\vbcc), as described elsewhere
  - you could then *cd* to the root folder *haujobb-amiga* and type *make* to build the whole Amiga side from scratch
- set up a shared folder on your PC or NAS that could be reached from your real or emulated Amiga, as described elsewhere
- installed Visual Studio, at least 2010, but probably the latest 2017 Community edition, as described elsewhere
- installed Qt5, Qt Creator, and cdb debugger support, as described elsewhere
- and then double-checked that compiler, debugger, and Qt version were correctly set up as a *kit*, as described elsewhere

This is all that needed to be done, so you could now open our supplied example project files (*.pro) in the */demo* folder and start making Amiga demos on PC. If you find any issues, please report them on GitHub.

Enjoy!

2 Assembler and Linker Setup

Traditionally, many Amiga-demos were made using ASM-One, or one of its derivates. I have personally worked with ASM-One v1.29 for many years, and demos like Mnemonics were made almost exclusively with that assembler, while other parts relied on genam from the Devpac 3.04 package. To honor that tradition, WickedOS stays fully compatible with ASM-One and Devpac. But, more importantly, it can also be used with tools like vasm and vlink, both natively on the Amiga, and for cross-compiling on the PC.

This setup chapter will take you on a quick tour through the different components that are required to get you up and running step by step.

To get started, grab the release archive of our framework from GitHub at https://github.com/leifo/haujobb-amiga.

2.1 Includes from NDK

First of all, you need to have the assembler include-files available at “INCLUDES:”. WickedOS doesn't need the latest features of AmigaOS, so it is likely to work with the includes that you might already have. If not, do as follows.

Obtain a copy of the latest AmigaOS Native Developer Kit (NDK). At the time of writing this was version 3.9 and it was still available at www.haage-partner.de/download/AmigaOS/NDK39.lha

1. Unpack the archive on your Amiga
2. Assign “INCLUDES:” <whereYouUnpackedIt>/Include/include_i
3. List includes:hardware/custom.i (should list a file, mine is 3045 bytes long)
4. Consider putting the assign from line 2 into s:user-startup (optional, but you must provide the assign to INCLUDES: if you plan to assemble WickedOS on Amiga).

2.2 vasm and vlink

We can now use *vasm* to assemble the WickedOS source into a linkable binary object (.o) file, and then link that into an executable file using *vlink*. These portable command-line tools are at the core of our strategy for cross-development, together with vmake and vbcc, which will be covered later in this document.

The project homepages are at http://sun.hasenbraten.de/vasm/and

http://sun.hasenbraten.de/vlink/. You can get the binaries, sources, as well as complete documentation for both tools at their respective homepages.

The supplied WickedOS source assembles out of the box on Amiga and PC. The following sections show you how to do it.

To assembe on the Amiga, get the latest release binaries from

- http://sun.hasenbraten.de/vasm/bin/rel/vasmm68k_mot_os3.lha
- and http://sun.hasenbraten.de/vlink/bin/rel/vlink_AmigaM68k.lha

Copy the contained files vasmm68k_mot and vlink to your path (e.g. to “C:”). Confirm that they are working by typing:

- vlink -v
- vasmm68k_mot

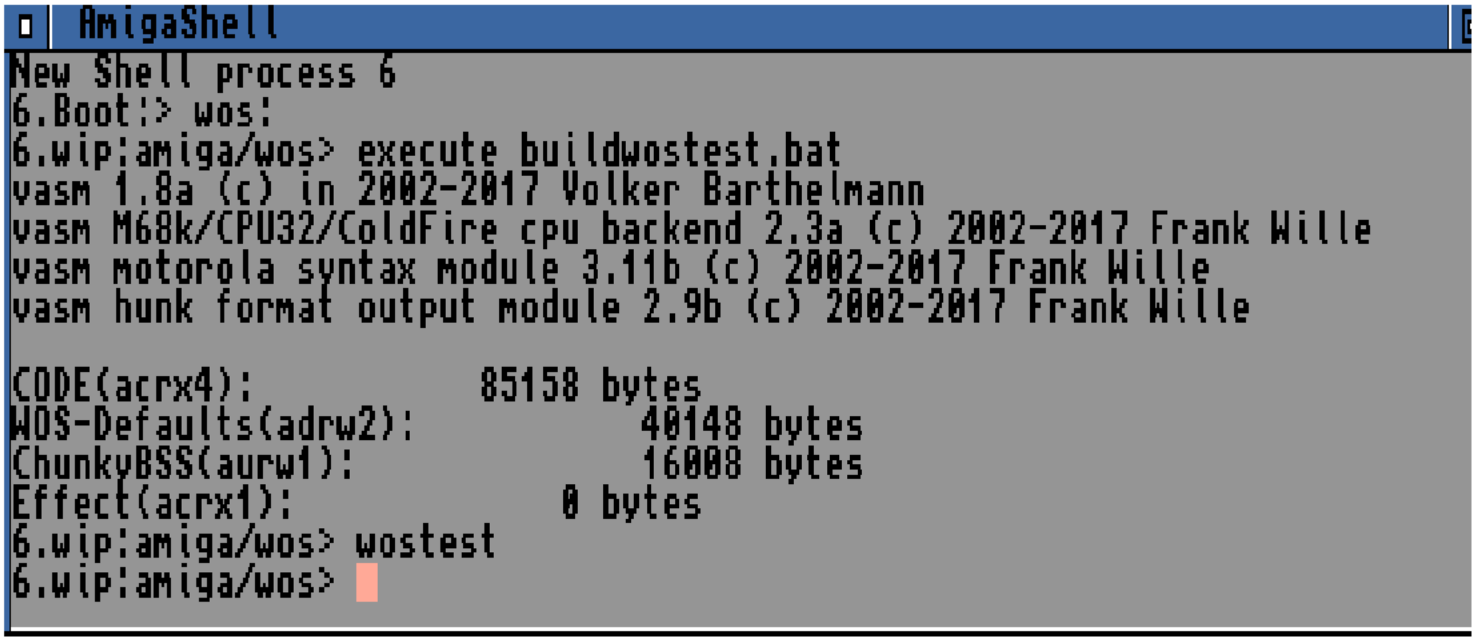
The output version numbers should somehow match those that are mentioned on the project homepages. Note that vasm is not only portable, i.e. ready to make work on different plattforms, but also retargetable, i.e. able to assemble sources for different target CPUs with different syntax modules. Shortcodes for the latter two are automatically appended to the binary executable name. Although many different versions of vasm exist, we are only interested in the 68k_mot version. If you are like me and like shorter names, you can add “alias vasm vasmm68k_mot” to your s:user-startup to make your life a little bit easier once in a while.

2.3 Example: Wostest

With vasm and vlink in place, you can already assemble, link and execute the built-in example like this:

1. cd WOS:
2. execute buildwostest.bat
3. wostest

This procedure should build and run an example screen. See the screenshot for a typical output of the shell and of *wostest*.



```
New Shell process 6
6.Boot:> wos:
6.wip:amiga/wos> execute buildwostest.bat
vasm 1.8a (c) in 2002-2017 Volker Barthelmann
vasm M68k/CPU32/ColdFire cpu backend 2.3a (c) 2002-2017 Frank Wille
vasm motorola syntax module 3.11b (c) 2002-2017 Frank Wille
vasm hunk format output module 2.9b (c) 2002-2017 Frank Wille

CODE(acrx4):            85158 bytes
WOS-Defaults(adrw2):            40148 bytes
ChunkyBSS(aurw1):               16008 bytes
Effect(acrx1):              0 bytes
6.wip:amiga/wos> wostest
6.wip:amiga/wos>
```

Figure 2.1: Wostest assembled on Amiga, output
vasm and vlink are called from buildwostest.bat, which contains only two lines. Let's look at them in more detail:

```
vasmm68k_mot -Fhunk -m68020 -m68882 wos_v1.63.s -o wostest.o
-Ic:/vbcc/NDK39/Include/include_i -DWTEST
vlink wostest.o -o wostest
```

The vasm-line specifies a number of things about our source and target binary via these parameters:

- -F: the output format should be (Amiga) hunk
- -m68020/m68882: the source contains 68020 CPU and 68882 FPU commands (this defines the base-level and is also okay if we really want to target 68060)
- -o: the assembled file should be written to a linkable object named wostest.o
- -D: Define a flag (here “WTEST”, which causes the conditional-assembly of the built-in example)
- -I: additional Include path at “c:/vbcc/targets/NDK39/Include/include_i” (as created by the Windows installer, c.f. elsewhere)
    - Note that this path is searched first, but not found on Amiga, since it is a PC-style path, and thus disregarded silently.
    - vasm thus falls back to using the “incdir INCLUDES:” directives that are contained in the source.
    - if you preferred, you could actually remove the assign INCLUDES: and point to the corresponding directory via the -I parameter.

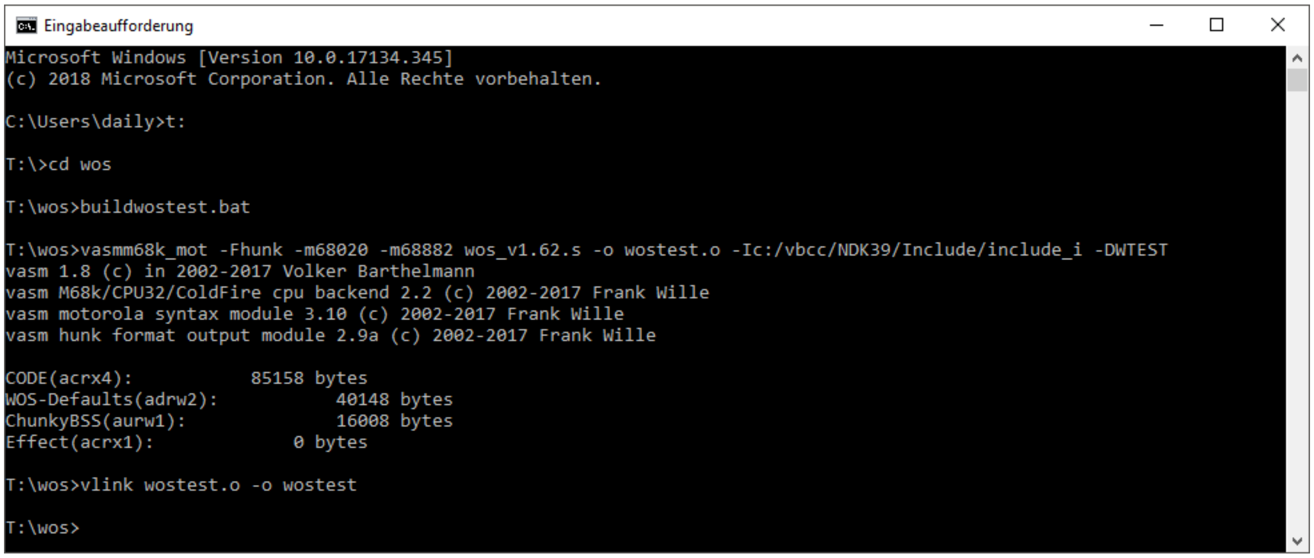
Finally, the vlink-line takes the output-file generated by vasm, wostest.o, and turns it into an executable file, *wostest*.

2.4 Assembling on PC

Windows binaries of vasm, vlink and make are shipped together with the vbcc-compiler which can be obtained from http://sun.hasenbraten.de/vbcc/

Download and run Install-VBCC.exe, as decribed elsewhere in more detail. Please skip to that page. Then come back and confirm that the tools are working by opening a command prompt and typing:

- vlink -v
- vasmm68k_mot



```
Microsoft Windows [Version 10.0.17134.345]
(c) 2018 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\daily>t:

T:\>cd wos

T:\wos>buildwostest.bat

T:\wos>vasmm68k_mot -Fhunk -m68020 -m68882 wos_v1.62.s -o wostest.o -Ic:/vbcc/NDK39/Include/include_i -DWTEST
vasm 1.8 (c) in 2002-2017 Volker Barthelmann
vasm M68k/CPU32/ColdFire cpu backend 2.2 (c) 2002-2017 Frank Wille
vasm motorola syntax module 3.10 (c) 2002-2017 Frank Wille
vasm hunk format output module 2.9a (c) 2002-2017 Frank Wille

CODE(acrx4):           85158 bytes
WOS-Defaults(adrw2):           40148 bytes
ChunkyBSS(aurw1):              16008 bytes
Effect(acrx1):              0 bytes

T:\wos>vlink wostest.o -o wostest

T:\wos>
```

Figure 2.2: Wostest assembled on PC
With vasmm68k_mot and vlink in place, you can then just:

1. cd <*to the directory where you unpacked WickedOS on PC*> (here: t:\wos)
2. type “buildwostest.bat” to cross-assemble and link *wostest* on the PC

Note that the binary output produced on the PC is exactly the same as on the Amiga, of course. Also note that this worked in a fraction of the time that it took on the Amiga. Yes, it was really *that fast*. Welcome to Modern Amiga Demo Cross-Development!

2.5 Shared Folders between Amiga and PC

Now that you cross-compiled an Amiga executable blazingly fast on your PC, you wouldn't want to waste any time getting it onto your Amiga for testing it, would you? So manual file-copying via FTP, CF-cards, or by other means is not going to be fast enough for quick turn-around times. Your ideal setup depends on whether you want to work with emulated or real Amigas, or both.

Emulators

like WinUAE allow adding a folder from the host-computer as a volume on the emulated Amiga. Do yourself a favour and use this feature!

I personally mounted a volume “WIP:” (as in Work In Progress) from a folder that I mounted from a Network Attached Storage (NAS, a Synology here) on the PC. This is because...

Real Amigas

cannot just mount your Windows folders. NAS folders, on the other hand, are made to be mountable, and can also easily be mounted on Amigas that have:

- some Fast-RAM
- a TCP/IP stack, like AmiTCP, Genesis, Miami, or Roadshow
- a SMB file system client like smbfs (try Aminet, although the version 1.74 hosted there is currently horribly outdated)
  - I am using Roadshow and just had to add the following line to s:Network-Startup

    ```
    smbfs domain=workgroup user=username password=password
    volume=NAS service=//diskstation/files/amiga
    ```

PC-Ramdisks

are a fact. Yes, they do exist! Even though they are not shipped as part of the Windows operating system, like on Amiga, you can easily download and install very powerful third-party components. And PC-based Ramdisks are just as useful as they have always been on Amigas. This is what I did:

1. Download and install ImDisk Virtual Disk Driver
2. Configure a Drive Letter “T:” with a 2 GB RamDisk using File System “NTFS”
   1. check: “Allocate Memory Dynamically”, “Launch at Windows Startup”
   2. uncheck: “Create TEMP Folder”
3. Mount that folder as “PCT:” in WinUAE (very useful for quick data exchange, esp. downloads)

3 Compiler Setup

When planning to develop Amiga software with the C programming language, there are a number of C-Compilers to choose from. Most notably they are:

- SAS/C, which is *the* Amiga standard programming environment (commercial and deprecated)
- GCC, which is *the* cross-plattform standard (free and open source)
- Storm C, an Amiga programming environment based on an old version of GCC (commercial)
- DICE, which was a popular and inexpensive Amiga programming environment (recently open sourced)
- vbcc, which is an actively maintained cross-plattform compiler (free and open source)

As we were looking for a cross-compiling solution, SAS/C, Storm C and DICE were no options. They are also neither free and open source, nor actively maintained, which was our second selection-criteria. This left us to decide between GCC and vbcc. At the time when we made our decision in 2009/2010, the Amiga-port of GCC was lacking behind in version numbers, while vbcc was the relatively new thing that was used by several Amiga-coders, and personally recommended by Kalms of TBL. It also allows for writing inline-assembly in the familiar Motoral syntax, rather than the more alien GCC Assembler syntax.

Moreover, vbcc is designed to work in combination with vasm and vlink from the same authors. The maintainer of the Amiga bindings, Frank Wille, is one of the most experienced and longest standing developers of quality Amiga development tools, ranging back to at least 1991 (with his PhxAssand PxhLnk). This long-term experience and continued commitment ensures first class Amiga support.

Hence, we decided to use vbcc as our C-compiler. The vbcc project homepage is located at http://sun.hasenbraten.de/vbcc/. There you can get the binaries, targets, sources, and complete documentation.

3.1 vbcc setup on Amiga

Installation on Amiga is easy. Citing the vbcc download instructions:

> You need to pick the appropriate binary archive for your host platform. Then you can add as many target archives as you need. [..] Install the binary archive first, using the provided Amiga installer, then add the targets.

So, download and install:

1. AmigaOS 2.x/3.x 68020+ binaries
2. Compiler target AmigaOS 2.x/3.x M680x0

The archives contain standard Amiga Installer scripts. Run them. Then test if the following commands are available:

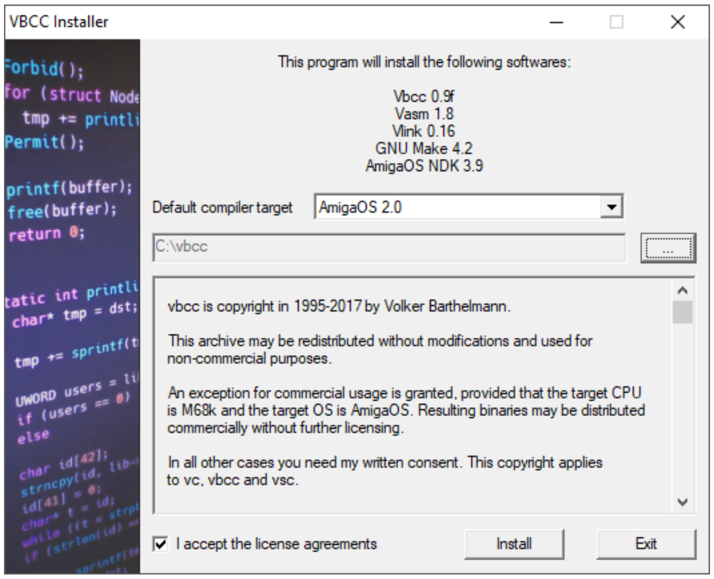
1. vc
2. vbccm68k

The former is a frontend for the latter and is very handy, if you are planning to use the C-compiler on the Amiga..

3.2 vbcc setup on PC

Installation on PC is even easier. Basically, just download and install a single file from the vbcc homepage. But be aware that the download is falsely reported to be a infected by malware by several virus scanners! Therefore:

1. Disable your virus-checker (optional)
2. Download and install Install-VBCC.exe with these settings
   1. Leave "Default compiler target" as *AmigaOS 2.0*
   2. Change path (below) to C:\vbcc



VBCC Installer

This program will install the following softwares:

Vbcc 0.9f
Vasm 1.8
Vlink 0.16
GNU Make 4.2
AmigaOS NDK 3.9

Default compiler target: AmigaOS 2.0

C:\vbcc

vbcc is copyright in 1995-2017 by Volker Barthelmann.

This archive may be redistributed without modifications and used for non-commercial purposes.

An exception for commercial usage is granted, provided that the target CPU is M68k and the target OS is AmigaOS. Resulting binaries may be distributed commercially without further licensing.

In all other cases you need my written consent. This copyright applies to vc, vbcc and vsc.

☑ I accept the license agreements

Figure 3.1: VBCC Installer for Windows

3. Enable your virus-checker again. If your were curious about this installer, you could:
   1. check the sources at GitHub
   2. follow the thread Windows installer for VBCC tool chain at the English Amiga Board
   3. or believe us that it is fine.

The good news is that you are already done! Open a new command prompt and test that the following commands are available:

1. vc
2. vbccm68k
3. vasmm68k_mot
4. vlink
5. make

3.3 Example: Hello world

Let's compile our first C-source with this setup! The following procedure should work on the Amiga aas well as on the PC side.

1. cd to the /demo/helloworld folder
2. vc hello.c
   1. this should generate an Amiga executable "a.out"
   2. alternatively provide a filename with: vc hello.c -o hello
3. Run the freshly compiled executable on Amiga.

For completeness, this is the full source of hello.c

```
/* hello world, noname, 23.10.18
compile with:
- "vc hello.c" (compiles to file a.out)
- or "vc +aos68k hello.c -o hello" (being more explicit about what we want)
- or "vc +aos68k hello.c -o hello -v" (to see commands as called by vc)
- or "make" (using makefile)
```

```
*/
#include <stdio.h>
int main(int argc, char **argv)
{
  int i;
  if (argc==1)
  {
    /* no args */
    printf("Hello, world!\n"); /* argv[0] is always the filename */
    printf("Called from filename: %s\n", argv[0]);
  }else{
    /* arguments given, start at argv[1] */
    printf("Hello");
    for (i=1; i<argc; i++)
    {
      printf(", %s",argv[i]);
    }
    printf("!\n");
  }
  return 0;
}
```

This is slightly extended hello world that also makes use of the provided arguments (i.e. argc and argv). You can also build the binary by executing build.bat. This example also comes with a simple makefile. But in order to build it from there, you first have to install make itself.

4 GNU make

While using batch-files for scripting repetetive tasks is useful, you are quickly hitting the roof with that approach. Which files need to be built, and how? This is where makefiles and make come into play. They coordinate the software build process in a structured way. Required tools, source-files, and libraries are listed in the makefile. Based on the rules that are also noted in the makefile, the make-tool then coordinates the build-process, i.e. turning source-code into an executable file.

We are using GNU make for building the Amiga target, as it is a free standard tool that is available on both Amiga and current plattforms. It is your choice to install it on Amiga and/or PC. The forthfollowing test should run on both platforms.

4.1 Configuring make on Amiga

1. Get a native Amiga binary from: http://aminet.net/package/dev/c/make-3.75-bin and put it to your path
    1. Please note that the v3.75 from Aminet is in fact a v3.74 ported natively for Amiga. The author just bumped the version number for his efforts.
    2. Open a newshell and type “make -v” to check that make is working
        1. it should show you its version number.
        2. In case you get a stack overflow warning type “stack 20000” and try again

4.2 Configuring make on PC

1. Make is bundled and installed with Install-VBCC.exe, as describedelsewhere (v4.2.90 at the time of writing). Thus you should have already installed it.
    1. This is the preferred version
    2. Alternatively, older Windows binaries can be obtained from: http://gnuwin32.sourceforge.net/packages/make.htm
2. Open a command prompt and type “make -v” to check that it is working
    1. make should show you its version number.

4.3 Building the WickedOS C2P-converters

We are aiming to use make on both Amiga and PC. Test that it works for you in two steps:

1. Use it to assemble a bunch of real world assembler files with a given makefile
    1. cd to /sub
    2. make

This should call vasm on about two dozen assembly-files from the /chunky subfolder and generate the same number of *.bin files. These files are the actual chunky to planar (C2P) converters for WickedOS' different screenmodes. If the distribution archive came only with one bin-file (for screenmode #1, as used in the initial example), you have now enabled the rest of them. Congratulations, well done!

4.4 Cleaning with rm

After work comes the clean-up! For make this is “make clean” which usually deletes a bunch of assembled or compiled binary object files and needs a delete command in order to work. Unfortunately, these delete commands differ between operating systems, not only by name, but also by syntax. In order to keep at least a somehow coherent structure, we define as per convention that a command “rm” should be reachable and work like rm from GNU coreutils.

4.4.1 On Amiga

Just add another line to s:user-startup: “alias rm delete”. Please note that this is not ideal but will work reasonably well for small projects, as long as the command-line does not get too long. Unfortunately, the command-line of the C2P-converter example with over 20 entries is already exceeding those limits. Mea culpa!

4.4.2 On PC

Windows del is not good enough for our purposes! Therefore we are using rm from GNU coreutils, a.k.a. fileutils.

1. Get the latest binaries from http://gnuwin32.sourceforge.net/packages/coreutils.htm
    1. either choose the setup file, or just the binary.zip
2. At your discretion, install all the tools, or just copy rm.exe from the bin-folder in the zip-file to your path
    1. it is suggested to copy rm to c:\vbcc\bin
    2. in case you choose the file from the zip-file, add missing dependencies from coreutils-dep-zip
3. Open a command prompt and type “rm --help” to check if it is working
4. Finally, try it in combination with make
    1. cd to the /sub directory again
    2. type “make” and it should tell you “Nothing to be done for `all'” which means that all assembled filed were up-to-date.
    3. type “make clean” to remove those files

4. type “make” again and see it rebuild.

This concludes the setup of the Amiga-oriented tools.

5 PC-Prototyping Environment

A large part of the productivity-benefits of our toolchain stem from the use of a native PC-build while prototyping the effects and editing the demo. This takes cross-compilation out of the picture for a period of time and allows you to focus on getting something on screen, debugging it, and then making it look nice before focussing again on making it all run smoothly on the Amiga. It also allows you to exploit the PC-hardware that you own anyway for your own sanity, e.g. use your big monitor wisely and fill it with a modern IDE with syntax-coloring, code-completion, code-navigation, project support, and of course debugging facilities. And run all of this natively on your modern multi-core CPU to save time and keep you in the flow.

For this to work you need a compiler, an IDE, and a debugger. While in theory these components would all be included in the popular Microsoft Visual Studio, we have blended it in our toolchain with QtCreator as the IDE component. While this was mainly a simple personal preference for a nice IDE that can work with different compilers, it also kept our toolchain itself plattform independant! QtCreator is available for Windows, macOS, and Linux. For the latter two, you would be able to set it up with gcc. For Windows, you best use it in combination with Microsoft's compiler.

5.1 Visual Studio

Let's start with installing Microsoft Visual Studio (MSVC), which comes in many different versions over the years (see table). The oldest one that we have personally tested is 2010. If you already have Visual Studio installed in a least that version, you can skip this section.

Visual Studio	2010	2012	2013	2015	2017
Pro	OK	not tested	OK	not tested	not tested
Community	-	-	not tested	OK	OK

Table 1: Supported Microsoft Visual Studio versions

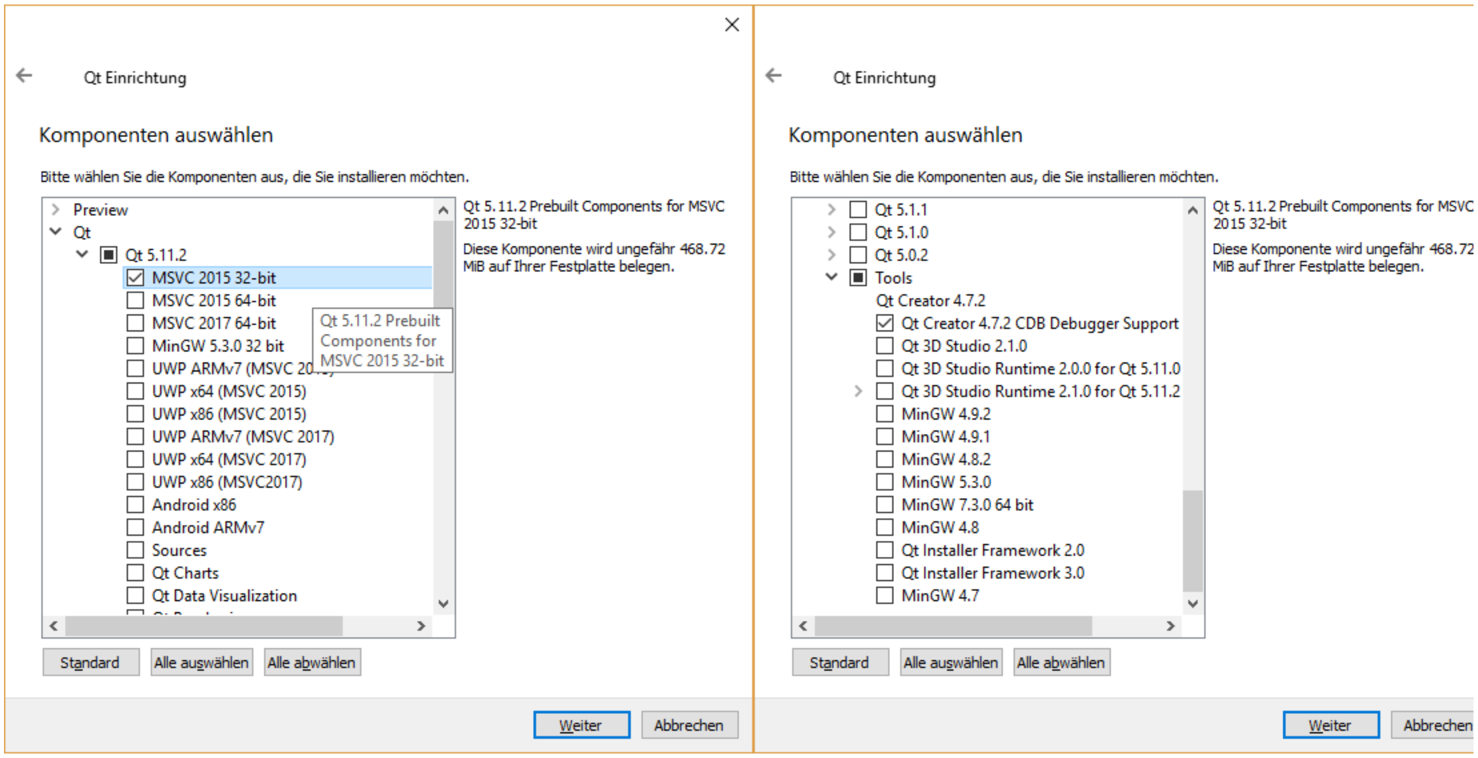
While Visual Studio was, and still is, a full-price commercial product, Microsoft has launched a number of free (as in beer) editions over the years. The Express edition introduced in 2005 has long since been discontinued and is now superseded by the Community edition. Visual Studio Community 2017 will work well for us. Please download it from https://visualstudio.microsoft.com and then install it on your Windows machine.

With Visual Studio Community 2017, please also install the missing standard headers by selecting *Windows Universal CRT SDK* under *Individual Components -> Compiler, Buildtools and Runtimes*, as described (9.1).

5.2 Qt Creator

Now let't install Qt and Qt Creator. The Haujobb Amiga Framework is known to work with Qt4 and Visual Studio 2010 as the minimum supported versions. If you haven't got Qt installed already, head over to https://www.qt.io/download, get the Open Source edition installer for Qt5, and run it.

It features many different options, we don't need all of them. Just install the latest available Qt 5.x.y release, as well as QtCreator and CDB support from the tools section (see figure 5.1).



Komponenten auswählen

Bitte wählen Sie die Komponenten aus, die Sie installieren möchten.

- Preview
- Qt
  - ■ Qt 5.11.2
    - ☑ MSVC 2015 32-bit
    - ☐ MSVC 2015 64-bit
    - ☐ MSVC 2017 64-bit
    - ☐ MinGW 5.3.0 32 bit
    - ☐ UWP ARMv7 (MSVC 20
    - ☐ UWP x64 (MSVC 2015)
    - ☐ UWP x86 (MSVC 2015)
    - ☐ UWP ARMv7 (MSVC 2017)
    - ☐ UWP x64 (MSVC 2017)
    - ☐ UWP x86 (MSVC2017)
    - ☐ Android x86
    - ☐ Android ARMv7
    - ☐ Sources
    - ☐ Qt Charts
    - ☐ Qt Data Visualization

Qt 5.11.2 Prebuilt Components for MSVC 2015 32-bit

Qt 5.11.2 Prebuilt Components for MSVC 2015 32-bit

Diese Komponente wird ungefähr 468.72 MiB auf Ihrer Festplatte belegen.

Komponenten auswählen

Bitte wählen Sie die Komponenten aus, die Sie installieren möchten.

- ☐ Qt 5.1.1
- ☐ Qt 5.1.0
- ☐ Qt 5.0.2
- ■ Tools
  - Qt Creator 4.7.2
  - ☑ Qt Creator 4.7.2 CDB Debugger Support
  - ☐ Qt 3D Studio 2.1.0
  - ☐ Qt 3D Studio Runtime 2.0.0 for Qt 5.11.0
  - ☐ Qt 3D Studio Runtime 2.1.0 for Qt 5.11.2
  - ☐ MinGW 4.9.2
  - ☐ MinGW 4.9.1
  - ☐ MinGW 4.8.2
  - ☐ MinGW 5.3.0
  - ☐ MinGW 7.3.0 64 bit
  - ☐ MinGW 4.8
  - ☐ Qt Installer Framework 2.0
  - ☐ Qt Installer Framework 3.0
  - ☐ MinGW 4.7

Qt 5.11.2 Prebuilt Components for MSVC 2015 32-bit

Diese Komponente wird ungefähr 468.72 MiB auf Ihrer Festplatte belegen.

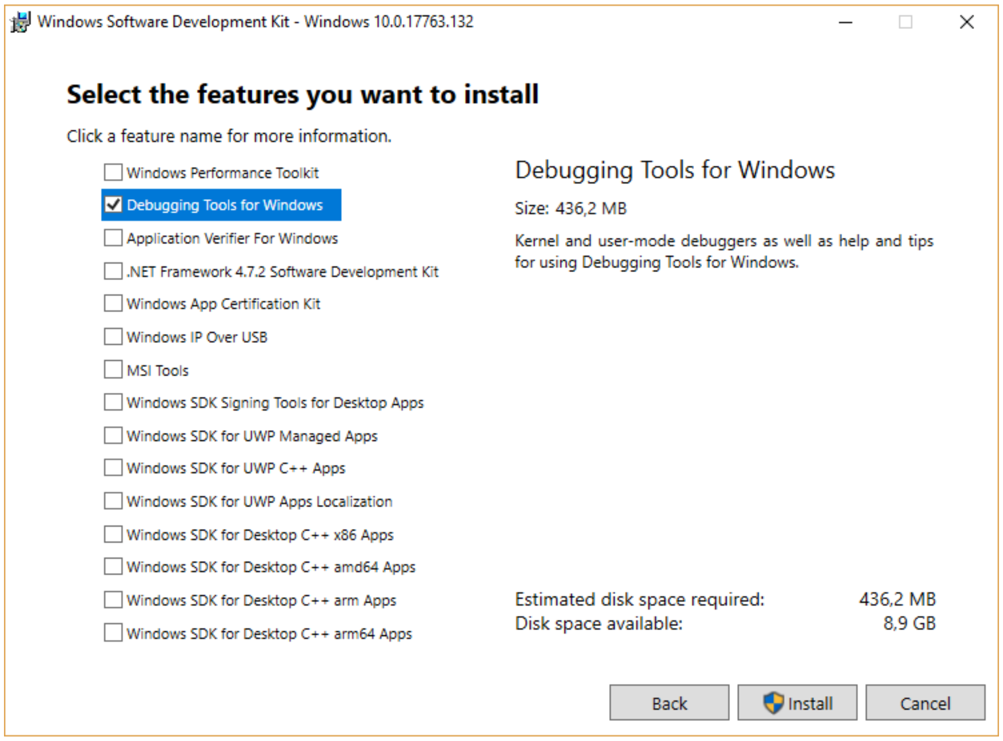
Figure 5.1: Required Components from Qt Open Source Installer

Please note that the version numbers in the figure will change quickly and are already outdated at the time of writing. The figure also shows a setup wizard that was run on a machine with MSVC 2015 32-bit. You will need to pick the version that fits your installed compiler. It is your choice, to install 32-bit or 64-bit version (if available). The default path for installation if C:\Qt.

The official documentation for Qt Creator is at http://doc.qt.io/qtcreator/.

5.3 Debugger (CDB)

If you want to debug your C-programs in Qt Creator, you need to setup the CDB debugger. The official documentation for this is here. It boils down to downloading the Windows SDK from https://developer.microsoft.com/de-de/windows/downloads/windows-10-sdk. Then run the installer and uncheck everything except “Debugging Tools for Windows” (see figure 5.2).


Windows Software Development Kit - Windows 10.0.17763.132

Select the features you want to install

Click a feature name for more information.

- [ ] Windows Performance Toolkit
- [✓] Debugging Tools for Windows
- [ ] Application Verifier For Windows
- [ ] .NET Framework 4.7.2 Software Development Kit
- [ ] Windows App Certification Kit
- [ ] Windows IP Over USB
- [ ] MSI Tools
- [ ] Windows SDK Signing Tools for Desktop Apps
- [ ] Windows SDK for UWP Managed Apps
- [ ] Windows SDK for UWP C++ Apps
- [ ] Windows SDK for UWP Apps Localization
- [ ] Windows SDK for Desktop C++ x86 Apps
- [ ] Windows SDK for Desktop C++ amd64 Apps
- [ ] Windows SDK for Desktop C++ arm Apps
- [ ] Windows SDK for Desktop C++ arm64 Apps

Debugging Tools for Windows

Size: 436,2 MB

Kernel and user-mode debuggers as well as help and tips for using Debugging Tools for Windows.

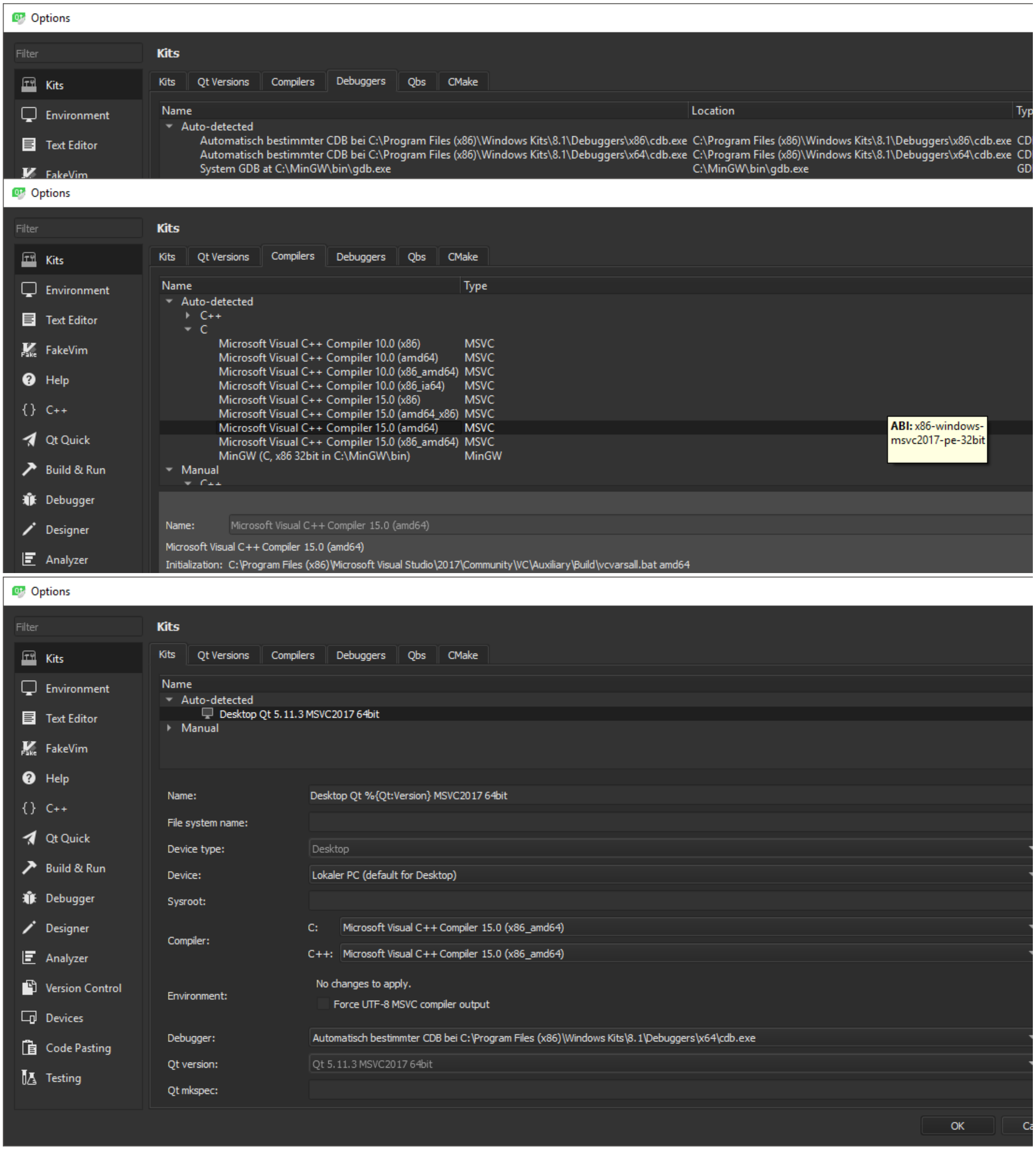
Estimated disk space required: 436,2 MB
Disk space available: 8,9 GB

Figure 5.2: Debugging Tools for Windows from Windows SDK

5.4 Kit Setup

In Qt Creator, combinations of compiler, debugger, Qt version, and a few other things are called a *kit*.

We need such a kit in order to work with our supplied project files (stars.pro, etc.). The full Qt doc for this is at http://doc.qt.io/qtcreator/creator-configuring-projects.html.



```
Name                                                                              Location                                                        Typ
▾ Auto-detected
    Automatisch bestimmter CDB bei C:\Program Files (x86)\Windows Kits\8.1\Debuggers\x86\cdb.exe  C:\Program Files (x86)\Windows Kits\8.1\Debuggers\x86\cdb.exe  CD
    Automatisch bestimmter CDB bei C:\Program Files (x86)\Windows Kits\8.1\Debuggers\x64\cdb.exe  C:\Program Files (x86)\Windows Kits\8.1\Debuggers\x64\cdb.exe  CD
    System GDB at C:\MinGW\bin\gdb.exe                                            C:\MinGW\bin\gdb.exe                                            GD
```

```
Name                                              Type
▾ Auto-detected
  ▸ C++
  ▾ C
      Microsoft Visual C++ Compiler 10.0 (x86)          MSVC
      Microsoft Visual C++ Compiler 10.0 (amd64)        MSVC
      Microsoft Visual C++ Compiler 10.0 (x86_amd64)    MSVC
      Microsoft Visual C++ Compiler 10.0 (x86_ia64)     MSVC
      Microsoft Visual C++ Compiler 15.0 (x86)          MSVC
      Microsoft Visual C++ Compiler 15.0 (amd64_x86)    MSVC
      Microsoft Visual C++ Compiler 15.0 (amd64)        MSVC      ABI: x86-windows-msvc2017-pe-32bit
      Microsoft Visual C++ Compiler 15.0 (x86_amd64)    MSVC
      MinGW (C, x86 32bit in C:\MinGW\bin)              MinGW
▾ Manual
  ▾ C++

Name: Microsoft Visual C++ Compiler 15.0 (amd64)
Microsoft Visual C++ Compiler 15.0 (amd64)
Initialization: C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Auxiliary\Build\vcvarsall.bat amd64
```

```
Name
▾ Auto-detected
    Desktop Qt 5.11.3 MSVC2017 64bit
▸ Manual

Name:              Desktop Qt %{Qt:Version} MSVC2017 64bit
File system name:
Device type:       Desktop
Device:            Lokaler PC (default for Desktop)
Sysroot:
Compiler:          C:   Microsoft Visual C++ Compiler 15.0 (x86_amd64)
                   C++: Microsoft Visual C++ Compiler 15.0 (x86_amd64)
Environment:       No changes to apply.
                   Force UTF-8 MSVC compiler output
Debugger:          Automatisch bestimmter CDB bei C:\Program Files (x86)\Windows Kits\8.1\Debuggers\x64\cdb.exe
Qt version:        Qt 5.11.3 MSVC2017 64bit
Qt mkspec:
```

Figure 5.3: Compiler-, Debugger-, and Kit-Settings in Qt Creator 4.7
When you installed compiler, debugger and Qt as described above, go to menu Tools->Options. The presented options windows should resemble figure 5.3 (note that it has been taken with an older Windows SDK installed).

Select *Kits* on the left pane. Then:

- check under *Qt Versions* that your installed Qt Version got auto-detected. If not, configure it manually.
- check under *Debuggers* that two versions of cdb.exe (32/64-bit) were found in the Windows Kits folder.
- check under *Compilers* that the Visual C++ compiler was correctly auto-detected. Note that the list will usually show several versions ending on something like “(amd64_x86)” meaning that this would be running on a 64-bit machine and generating code for a 32-bit machine. It usually doesn't matter to our framework if you are selecting a 32-bit version or not, as we are not going anywhere near the 32-bit limits, anyway.
- check under *Kits* that you have an auto-detected kit comprising of a *Qt Version*, a *Debugger*, and a *Compiler*.
- close the Options window, and you should be able to build & run, as well as to debug.

- keep in mind that sometimes you want to
    - run qmake by right-clicking root-element of a project-tree and selecting *Run qmake*
    - and then selecting *Rebuild*

5.5 Add-Ons

This section contains a few useful add-ons. Install them at your own discretion.

5.5.1 Syntax Highlighting for M68000 Assembly

This is really handy if you often work with M68000 assembly files in Qt Creator. Get asm-m68k.xml and install it. The documentation of *Generic Highlighting* in Qt Creator, as described in http://doc.qt.io/qtcreator/creator-highlighting.html should serve you as a basis. On my machine, I just copied the file to the *Fallback Location* (here: C:\QtSdk\Tools\QtCreator\share\qtcreator\generic-highlighter).

5.5.2 Qt Plug-In for Visual Studio

If you'd like to use our supplied Qt Creator Project files (*.pro) from within Visual Studio, you can install the *Qt Visual Studio Tools* as described in this blog-postand provided here(for 2015) and here(for 2017). You can also install it via the *marketplace* of Visual Studio.

Once installed:

1. Select menu *Qt VS Tools-> Qt Options*
    1. add your Qt version
        1. the name is automatically provided
        2. just select the path that leads to /bin/qmake under c:\Qt
        3. this only works for Qt5 and up
2. Select menu *Qt VS Tools->Open Qt Project File (.pro)...*
3. Build & Run

Please note that our IDE of choice is Qt Creator, so if you run into problems with Visual Studio, we would be more happy to hear about your solutions, and less prepared to help you fix it.

5.5.3 Symbol Files

Even though you have already installed a lot of data, the symbol files have not been part of it. They would usually be loaded on demand over the Internet connection and then cached locally. If you want to prepare yourself for longer times without Internet, get the symbol files. You can read more about this topic in this article. We also have a direct download link somewhere and will add it here at some point.

Part II Demomaking

6 Single Examples

This chapter takes you through the provided examples in order of complexity.

6.1 Hello World

Let's start with a simple hello world. We just want to build and run it from within Qt Creator to confirm that everything works.

6.1.1 Overview

- Open Qt Creator
- Select menu *File -> Open File or Project ... (CTRL-O)*
    - Open *helloworld.pro* from the */demo/helloworld* folder
    - alternatively, you can also double-click on a project (*.pro) file in Windows Explorer
- Upon first start of a project in Qt Creator, select your previously configured kit and press *Configure Project*
- Select menu *Build -> Run (CTRL-R)*

You should be presented with a console window and some printed text.

6.1.2 Discussion

It couldn't get much simpler than this. Basically, just a couple of printf()'s, which require stdio.h. If compilation fails for you on that header file, solve the issue now, as described elsewhere.

The example also shows the use of parameters that are provided to *int main(int argc, char **argv)*, with *argc* being the argument count, *argv[0]* the filename, and *argv[1..n]* the supplied arguments, if *argc*>1.

Finally, the project also showcases the need for a shadow-build-subdirectory (such as */build*), as Qt Creator calls qmake which creates its own “makefile”-file and thereby overwrites our existing “makefile” in GNU make syntax (9.1), which we need to build for Amiga. All further examples will thus employ a */build* folder for the Amiga-specific build files.

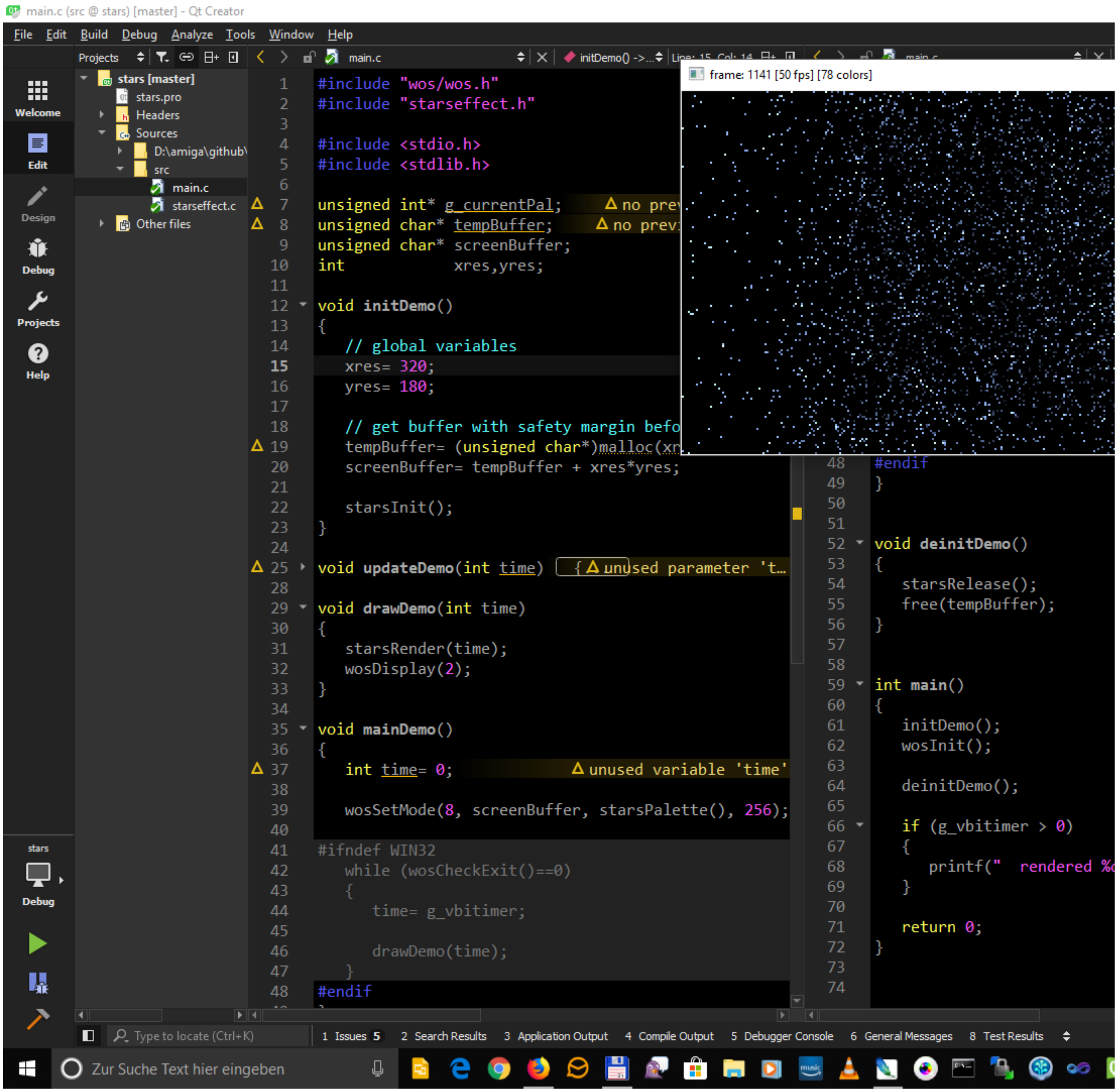
6.2 Stars

Just a classic star-field.

6.2.1 Overview

- Open, configure, build and run *stars.pro* from the */demo/stars* folder

You should be presented with a graphics window and some shaded 3D stars flying towards you, as show in figure 6.1.



frame: 1141 [50 fps] [78 colors]

```
1   #include "wos/wos.h"
2   #include "starseffect.h"
3
4   #include <stdio.h>
5   #include <stdlib.h>
6
7   unsigned int* g_currentPal;     ⚠ no pre
8   unsigned char* tempBuffer;     ⚠ no prev
9   unsigned char* screenBuffer;
10  int            xres,yres;
11
12  void initDemo()
13  {
14     // global variables
15     xres= 320;
16     yres= 180;
17
18     // get buffer with safety margin befo
19     tempBuffer= (unsigned char*)malloc(xr
20     screenBuffer= tempBuffer + xres*yres;
21
22     starsInit();
23  }
24
25  void updateDemo(int time)  { ⚠ unused parameter 't…
28
29  void drawDemo(int time)
30  {
31     starsRender(time);
32     wosDisplay(2);
33  }
34
35  void mainDemo()
36  {
37     int time= 0;           ⚠ unused variable 'time'
38
39     wosSetMode(8, screenBuffer, starsPalette(), 256);
40
41  #ifndef WIN32
42     while (wosCheckExit()==0)
43     {
44        time= g_vbitimer;
45
46        drawDemo(time);
47     }
48  #endif
```

```
48  #endif
49  }
50
51
52  void deinitDemo()
53  {
54     starsRelease();
55     free(tempBuffer);
56  }
57
58
59  int main()
60  {
61     initDemo();
62     wosInit();
63
64     deinitDemo();
65
66     if (g_vbitimer > 0)
67     {
68       printf("  rendered %
69     }
70
71     return 0;
72  }
73
74
```

Figure 6.1: Qt Creator session with project stars

6.2.2 Discussion

This example does not load any external files. The starfield is set up and drawn from a seperate file called starseffect.c (with fitting header starseffect.h) which is then called from main.c. Keeping the effect-code seperated like this is good practice and will become very useful once we start putting together a demo.

Let's study the code in figure 6.1starting from main():

- initDemo() is called first thing
    - this declares screensize global variables xres=320 and yres=180
    - it allocates memory for a screenBuffer (also global variable)
    - and calls starsInit()
        - this is obviously our effect init-code
        - made known via *#include “starseffect.h”* in line 2
    - starsInit() is not entirely shown in the figure
        - it calculates quasi random 3D star-coordinates
        - and calculates a palette
- wosInit() is called next
    - this sets up the graphics display amongst other things, and might potentially disable multi-tasking on the Amiga (depending in wether or not wickedquicklink.s / wickedlink.s have been assembled with or without the *KILLER* flag.)

  - wosInit() does not immediately return, but
    - hands over control to mainDemo()
    - only returns to main() once the demo is over
- mainDemo()
  - wosSetMode(8, screenBuffer, starsPalette(), 256);
    - sets up a low-res 320x180 screenmode (#8)
    - fed from screenBuffer
    - with colours from starsPalette() at normal brigthness (256)
  - on Amiga
    - mainDemo() then loops over drawDemo(), providing time as ticks (1/50 s)
    - until wosCheckExit() returns !=0
  - on PC (WIN32)
    - mainDemo() immediately returns
    - and drawDemo(time) is called from somewhere else from Qt
  - (this is why there is this *#ifndef WIN32 [..] #endif* block)
- drawDemo() just
  - renders the stars effect based on the given time
  - updated the display with wosDisplay(2);
    - the 2 denotes a lock to 25 fps maximum
    - which is realistic for low-res chunky to planar effects on AGA/060 config
    - motion will look smoother at a constant framerate rather than a slightly faster on
    - you could try (1) for a maximum 50 fps frame-lock or (0) for no lock
- starsRender()
  - clears the screenBuffer
  - calls starsDraw(), with
    - time
    - 32 as a constant factor for the perspective
    - screenbuffer, xres, yres
- starsDraw() finally
  - iterates over the number of stars (7000)
  - and for each star
    - apply z-movement based on time with boolean clipping
    - project x/y based on a floating point perspective factor derived from 1/z *variable factor
    - determines a pixel-colour based on z
    - adds that colour to the screenbuffer if the pixel is inside the x/y/z box
- upon exit, deinitDemo() is called, which
  - calls starsRelease() - an empty function/stub
  - frees the tempBuffer that contained the screenBuffer and some safety margin

The resulting effect is a classic, albeit simple starfield. It doesn't rotate or even just move in different directions. All of this could be added, if needs be. But we recommend you to go on and just take this as an example of a typical project setup.

6.2.3 Compile for and run on Amiga

If you now wanted to compile the stars for Amiga, you would:

- open a console window on PC
- *cd* to the */stars/build* folder
- run *make*

And then to run it on the Amiga, you would:

- *cd* to the */stars* folder on Amiga
- run *build/stars*

For this simple project, which doesn't load any external data-files, you could actually also run it from the *build* folder. But as soon as you want to load data (which we put in */data* by convention), you would need to make sure you run your executables from the right folder.

6.3 Picture

This example loads in and displays a picture that has been written in the standard GIF file-format.

6.3.1 Overview

- Open, configure, build and run *picture.pro* from the */demo/picture* folder
- You should be presented with a picture in 320x180 resolution and 8-bit colour-depth

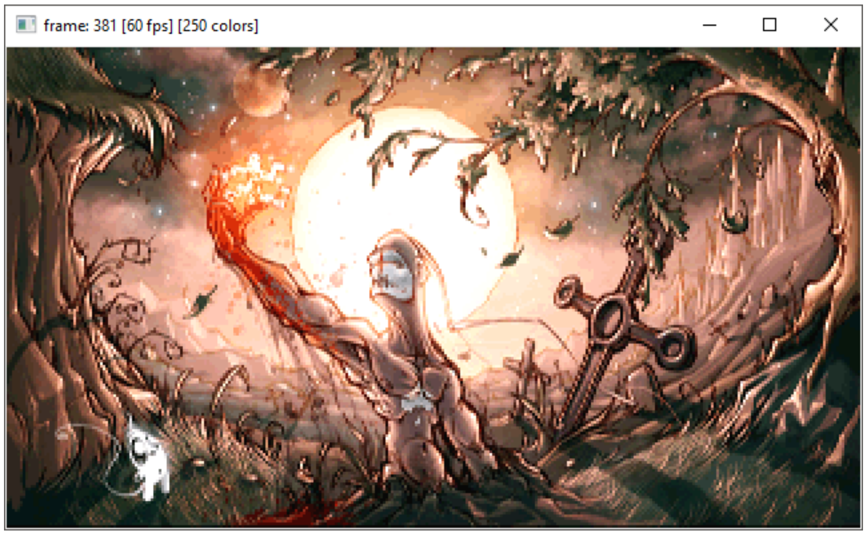


Figure 6.2: Picture project (artwork by Helge/Haujobb)

6.3.2 Discussion

The effect simply loads a picture in GIF-format which can be created with standard tools like Photoshop or Pro Motion. This is a lot easier and less time-consuming than always employing a graphics converter program.

```
void pictureEffectInit() {
int w,h;
gifLoad("data/helge-haujobb.gif", (void**)&picture, &w,&h,
(unsigned int*)&picturepal);
g_currentPal = picturepal;
}
```

The control flow of this example is the same as in the previous example stars. So upon init, the pictureEffectInit() function is called, which calls gifLoad() with

- the filename
- a pointer (&picture) to where the pointer to the loaded and decoded picture should be stored
    - note that GIF is an 8-bit format with up to 256 colours
    - picture is thus defined as unsigned char*
- pointers to the int variables w & h which will store width and height of the decoded image
- a pointer to an array of 256 unsigned int to store the image palette

Finally, pictureEffectInit() also copies the pointer to the loaded image palette to g_currentPal, which can be accessed from main.c in function drawDemo(). Here, the palette brightness will be calculated with a simple time-based function that makes the image fade in and out smoothly on a sine-wave. Note that the standard brightness for paletted image is 256. In this example it is just going slightly above that for an exaggerated effect.

6.3.3 Changing the palette

If you run this example on the Amiga (like the stars example), you will notice that the background colour around the image is not black and thus also fades in and out. This is often not the desired effect. To fix it, we need to have look at the palette.

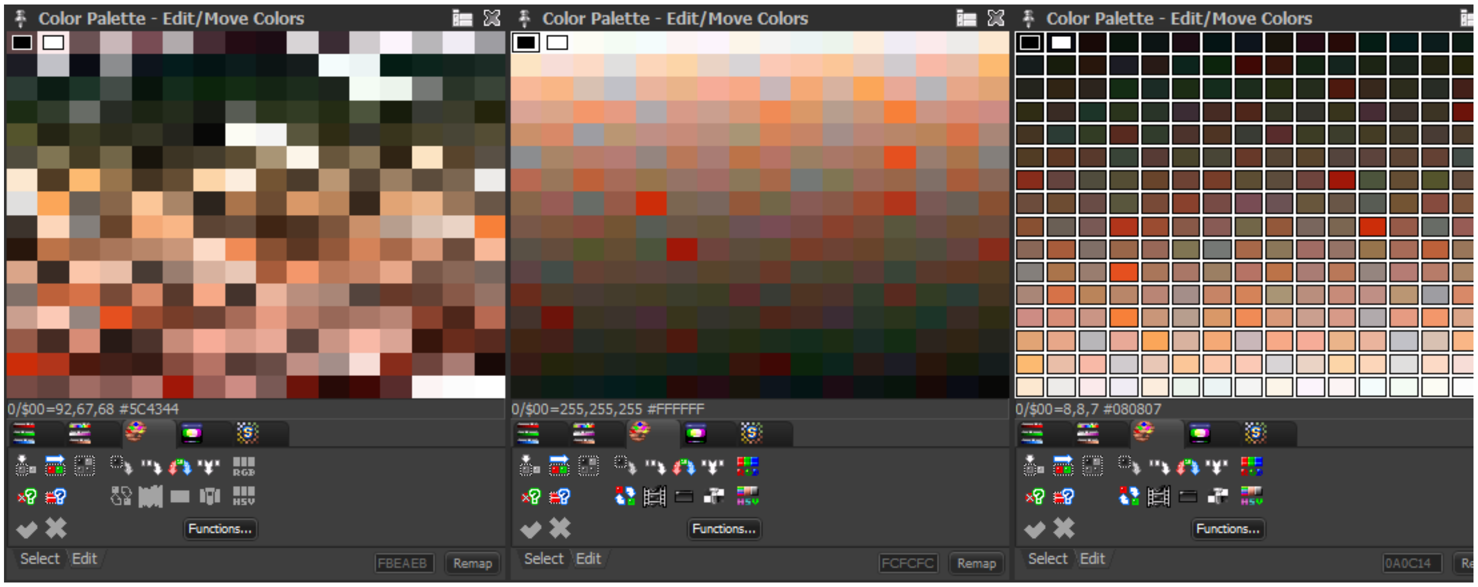


Figure 6.3: Sorting the picture palette with the Pro Motion NG palette editor

Tools like Photoshop are very good for all sorts of image manipulations and can also be used to save in the GIF-format. But it does not sort the colours in the way we would prefer it, i.e. keep colour 0 dark. Photoshops' palette editor is also below standard. Thus we will use a different PC-based tool to fiddle with the palette: *Pro Motion NG* from https://www.cosmigo.com/. The available free version is enough for this excercise. So download and install it, then:

- Open the image "data/helge-haujobb.gif" by selecting menu *File -> Load Image ...*
- You should see the image and the palette-editor (as in 6.3, left).
    - If not, press F12 to open it.

- Now press the *Functions* button in the palette editor and select *Sort -> Brightness*.
    - Ok, so the colors are sorted now, but from brightest to darkest. Your palette should look like in 6.3, middle.
    - But the image still looks odd.
- Press the *Remap* button in the palette editor, or select menu *Colors -> Resample (!) Color*
- To have then have the palette sorted from darkest to brightest we have to use flip function now *(Functions -> Flip*).
    - Your palette should look like in 6.3, right, as desired.
- Press *Remap*, again.

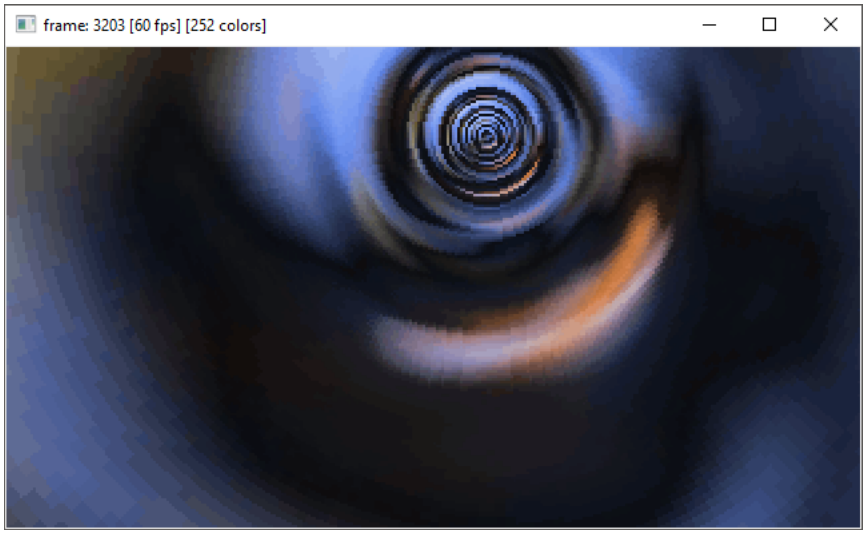
You can save the image, reload the Amiga executable without compiling it again, and the border should stay dark.

6.4 Movetable

This example shows a typical movetable effect in 320x180 pixel resolution and 8-bit colour-depth.

6.4.1 Overview

- Open, configure, build and run *movetable.pro* from the */demo/movetable* folder
- You should be presented with an effect in 320x180 resolution and 8-bit colour-depth



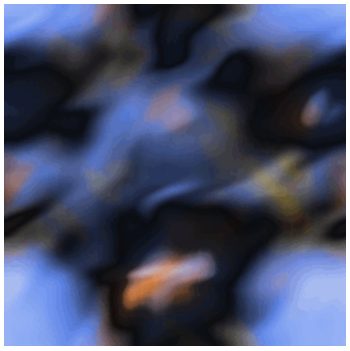


Figure 6.4: Movetable effect and underlying texture (texture by JCS/Haujobb)

6.4.2 Discussion

The effect uses a texture of size 256x256 which is loaded from a GIF file, like in the previous example. Furthermore, it uses a movetable which is generated using trigonometric functions and finally uses those two components to build the screen by obtaining per pixel texture coordinates from the movetable, moved by a smoothly animated x- and y-offset on a sine-wave. The movetable is exactly twice the size of the screen, so 640x360 in this case, and represents a tunnel shape. Other shapes are possible. Let's look at some details.

```
void movetableInit() {
int tabsize; int w,h;
gifLoad("data/texture.gif", (void**)&texture, &w,&h,
(unsigned int*)&texturepal);
// provide 256x256x8 tga-file and test this
//tgaLoad8("data/texture.tga", (void**)&texture, &w,&h,
(unsigned int*)&texturepal);
g_currentPal = texturepal;
// allocate memory
tabsize= tabx*taby;
// table size
xtab= (char*)malloc(tabsize);
ytab= (char*)malloc(tabsize);
// build tables
//buildSphere(xtab, ytab);
//buildWater(xtab, ytab);
buildTunnel(xtab, ytab);
}
```

If you look at function moveTableInit() from movetableeffect.c you will see how the texture is loaded and the function to build the table is called.

Just after the gifLoad() function is a tgaLoad8() function with the same signature, which loads the Targa Image File format TGA. Targa is supported by Photoshop and other tools. It is usually uncompressed and loads faster than GIF. The bigger file-size of TGA is compressed away when you bundle your demo in a compressed archive. Furthermore, Targa files also support higher colour depths; a feature that will come in handy when you work with true colour screenmodes. As an exercise, you could:

- provide a file called texture.tga in 256x256 pixel size and 8-bit colour depth
- comment out the gifLoad() line and comment in the tgaLoad8() line
- the effect should work as before

moveTableInit() then allocates some memory and calls buildTunnel(), one of three supplied build-functions, to fill that memory with movetable data.

- Try out the other two!

7 Demo Example

This chapter pulls all the strings together and provides a complete example demo that comprises of the previously discussed component with some added streaming music. The effects' parameters are controlled via the excellent Rocket sync-tracker. Although this approach to demo-syncing is quite common in the PC-scene for many years, it is still fairly unknown in the Amiga-scene and only used by very few demo-makers, like e.g. Loonies. In Haujobb, we have started using Rocket when developing the Beam Ridersdemo. We have previously spoken about our demo-making process in our Evoke 2018 talk Modern Amiga Demo Cross-Development (YouTube). With the release of our Amiga demo framework, we are now publicly sharing a complete and documented example of how to sync your Amiga demos with Rocket. This is how we did it. And now you can do it, too!

7.1 Hello Demo

7.1.1 Overview

- Open, configure, build and run *hellodemo.pro* from the */demo/hellodemo* folder
- The demo will refuse to start with the error message: “Could not connect to rocket editor!”
- This is because the source-code of hellodemo is configured
    - to require the Rocket editor on the PC
    - and to play the demo from the stored parameters on the Amiga.

7.1.2 SYNC_PLAYER

If you wanted to quickly check the example demo on the PC, you thus need to configure the source-code to not require the Rocket editor on the PC. This is done with the *SYNC_PLAYER* define in line 1 of main.c. If it is defined, the demo will load the **.rt* files from the /data directory and then play the demo with those parameters. So uncomment that define and run hellodemo again. It should start and show several effects timed to some music by Muffler.

When you are done with that, comment that define-line out again, as you usually would want to work in edit mode on PC.

Note: the Amiga build gets the SYNC_PLAYER define injected via the makefile (-DSYNC_PLAYER in CFLAGS, currently in line 40, c.f. section 2.1 in VBCC docs).

7.1.3 Good Practice Track Names

Rocket track names are your interface to your demo. For a typical demo, you will quickly have dozens of track names, so it is a Good Idea TM to name them properly. They also need to operate within certain limits:

- keep names short, meaningful and lowercase
- all Rocket tracks are Floats by default
    - but you can cast them e.g. to Int within your code
- Furthermore they are also saved to disk with filenames comprising of a certain scheme (we chose: sync_*.rt, where * is effectname_variablename, e.g. sync_movetable(2)_time.rt) that will have to fit within the Amiga Fast File System filename-length limit of 30 chars.

From our own experience we can propose the following:

- part : should be used as an Int to keep track of the currently visible part
- brightness : Int for the current palette brightness (if applicable), 0 means black, 256 is normal, >256 brighter
- effects should expose their name plus their part number in brackets, e.g. movetable(2), as some kind of namespace
    - effect variables are are appended to this namespace-name with a colon (“:”)
    - see the code from getSyncTracks() from main.c as an example

```
void getSyncTracks() {
   // rocket get tracks
   // general
   rt_part = sync_get_track(rocket, "part");
   rt_brightness = sync_get_track(rocket, "brightness");
   // pictures
   rt_picture_id = sync_get_track(rocket, "picture(0):id");
   // stars
   rt_star_time = sync_get_track(rocket, "stars(1):time");
   rt_star_pers = sync_get_track(rocket, "stars(1):perspective");
   // movetable
   rt_move_time = sync_get_track(rocket, "movetable(2):time");
   rt_move_xtab = sync_get_track(rocket, "movetable(2):xtab");
   rt_move_ytab = sync_get_track(rocket, "movetable(2):ytab");
}
```

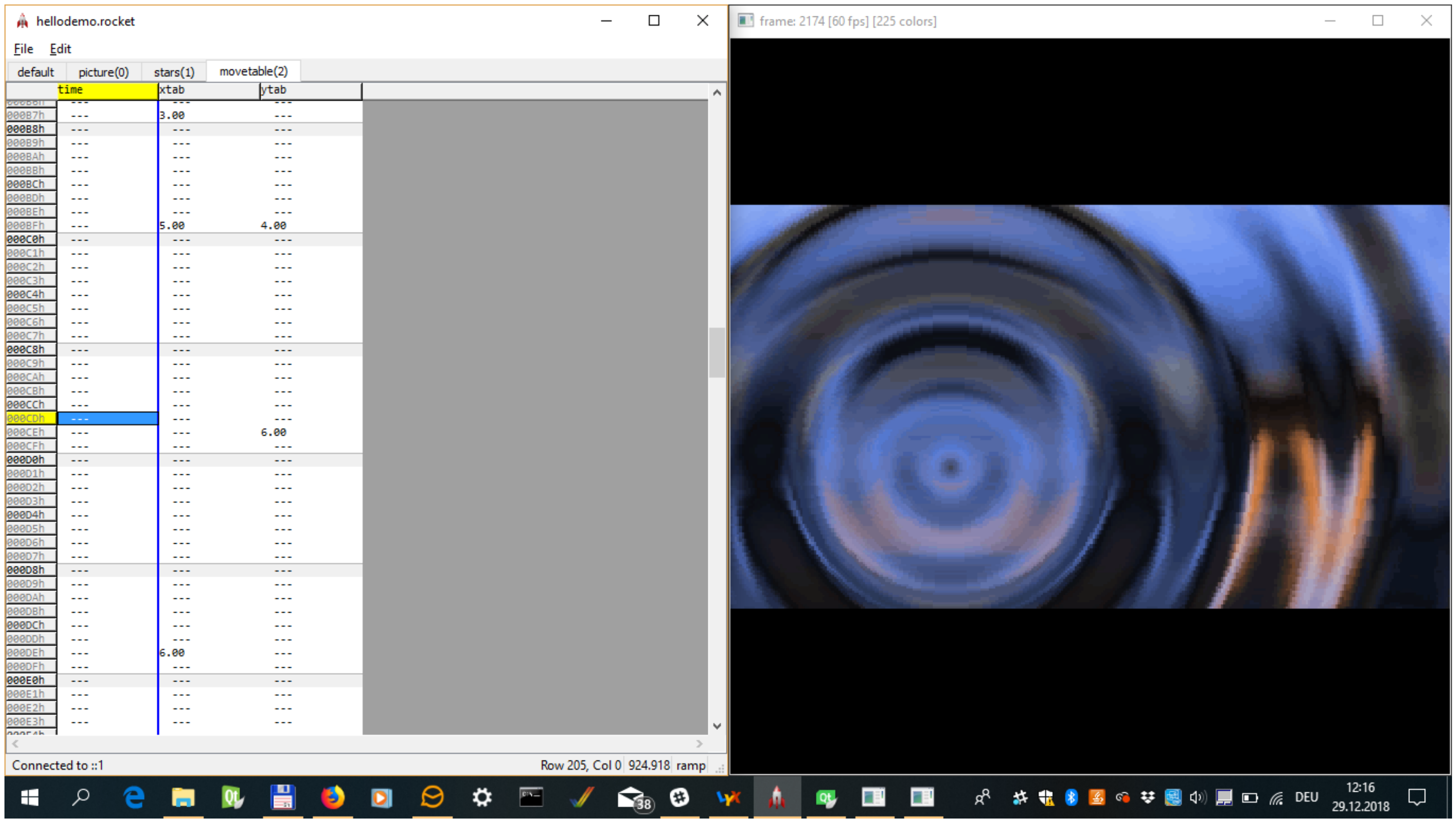
Adhering to such a scheme will make your life easier when wiring up the internals in your code and when timing your demo. So better stick to it.

7.2 Rocket Editor

Rocket is is an intuitive new way of... bah, whatever. It's a sync-tracker, a tool for synchronizing music and visuals in demoscene productions. It consists of a GUI editor (using Qt), and an ANSI C library that can either communicate with the editor over a network socket, or play back an exported data-set. – from Rocket GitHub page.

To get started with using the Rocket editor, download the latest official binaries from the GitHub Rocket releases. The archives come readily bundled with the required libraries, so just unpacking and starting *editor.exe* should work. Since its inception, several ports of the Rocket editor have appeared. All of them are listed on the Rocket GitHub page. We personally only worked with the original editor, but also quickly tested the alternative OpenGL editor and can confirm that it works.

On Windows, the Windows firewall would usually come up when starting your chosen editor for the first time. Just allow it to pass, as the editor will need to communicate with the demo via network.



```
         time       xtab       ytab
000B7h   ---        3.00       ---
000B8h   ---        ---        ---
000B9h   ---        ---        ---
000BAh   ---        ---        ---
000BBh   ---        ---        ---
000BCh   ---        ---        ---
000BDh   ---        ---        ---
000BEh   ---        ---        ---
000BFh   ---        5.00       4.00
000C0h   ---        ---        ---
000C1h   ---        ---        ---
000C2h   ---        ---        ---
000C3h   ---        ---        ---
000C4h   ---        ---        ---
000C5h   ---        ---        ---
000C6h   ---        ---        ---
000C7h   ---        ---        ---
000C8h   ---        ---        ---
000C9h   ---        ---        ---
000CAh   ---        ---        ---
000CBh   ---        ---        ---
000CCh   ---        ---        ---
000CDh   ---        ---        ---
000CEh   ---        ---        6.00
000CFh   ---        ---        ---
000D0h   ---        ---        ---
000D1h   ---        ---        ---
000D2h   ---        ---        ---
000D3h   ---        ---        ---
000D4h   ---        ---        ---
000D5h   ---        ---        ---
000D6h   ---        ---        ---
000D7h   ---        ---        ---
000D8h   ---        ---        ---
000D9h   ---        ---        ---
000DAh   ---        ---        ---
000DBh   ---        ---        ---
000DCh   ---        ---        ---
000DDh   ---        ---        ---
000DEh   ---        6.00       ---
000DFh   ---        ---        ---
000E0h   ---        ---        ---
000E1h   ---        ---        ---
000E2h   ---        ---        ---
000E3h   ---        ---        ---

Connected to ::1                         Row 205, Col 0  924.918  ramp
```

Figure 7.1: Rocket editor (left) and hellodemo (right)

Once you started the editor, you can recompile (without SYNC_PLAYER defined) and run the hellodemo project. You should immediately see some magic happen in the editor. As the demo and the editor already communicated about the available sync tracks, the editor should show them neatly organised in tabs, here: *default*, *picture(0)*, *stars(1)*, *movetable(2)*. The sync tracks still have no keyframes and interpolation methods assigned, yet. This is the typical state of how you would start with a clean slate once you wired up your Rocket variables in your code.

To get started with our example, load the provided file *hellodemo.rocket* from /data/hellodemo into your editor. This should populate the sync-tracks with data (see 7.1). You can now start and stop playing the demo by pressing <SPACE> in the editor.

7.2.1 Using the Editor

Note: This section is copied from the official Rocket documentation.

The Rocket editor is laid out like a music-tracker; tracks (or columns) and rows. Each track represents a separate "variable" in the demo, over the entire time-domain of the demo. Each row represents a specific point in time, and consists of a set of key frames. The key frames are interpolated over time according to their interpolation modes.

Interpolation modes

Each key frame has an interpolation mode associated with it, and that interpolation mode is valid until the next key frame is reached. The different interpolation modes are the following:

Step : This is the simplest mode, and always returns the key's value.

Linear : This does a linear interpolation between the current and the next key's values.

Smooth : This interpolates in a smooth fashion, the exact function is what is usually called "smoothstep". Do not confuse this mode with splines; this only interpolates smoothly between two different values, it does not try to calculate tangents or any such things.

Ramp : This is similar to "Linear", but additionally applies an exponentiation of the interpolation factor.

Keyboard shortcuts

Some of the Rocket editor's features are available through the menu and some keyboard shortcut. Here's a list of the supported keyboard shortcuts:

Shortcuts	*Action*
Up/Down/Left/Right	Move cursor
PgUp/PgDn	Move cursor 16 rows up/down
Home/End	Move cursor to begining/end
Ctrl+Left/Right	Move tracks
Enter	Enter key frame values
Del	Delete key frame
i	Enumerate interpolation mode
k	Toggle row-bookmark
Alt+PgUp/PgDn	Go to prev/next row-bookmark
Space	Pause/Resume demo
Shift+Up/Down/Left/Right	Select
Ctrl+C	Copy
Ctrl+V	Paste
Ctrl+Z	Undo
Shift+Ctrl+Z	Redo
Ctrl-B	Bias key frames
Shift+Ctrl+Up/Down	Quick-bias by +/- 0.1
Ctrl+Up/Down	Quick-bias by +/- 1
Ctrl+PgUp/PgDn	Quick-bias by +/- 10
Shift+Ctrl+PgUp/PgDn	Quick-bias by +/- 100

Table 2: Rocket editor shortcuts

7.3 Wiring up your variables to Rocket

Wiring up your code's internal variables to Rocket so that they can be modified over the network connection with the Rocket editor is at the heart of working with Rocket from a coder's perspective. It is a straight forward process that you will quickly embrace once you know what needs to be done.

All of this is usually happening in your demo's main.c - the effects do not know about Rocket at all. For the purpose of this write-up, we are just handling the case of the stars effect. All other effects are handled in the same way and you can follow everything in full detail in hellodemo's main.c.

7.3.1 Declarations

Include Rocket definitions from:

```
#include "../shared/rocket/lib/sync.h"
```

For working with Rocket, you always need a *sync_device*:

```
struct sync_device *rocket;
```

You would then define multiple *sync_tracks*, one for every variable that you want to animate:

```
const struct sync_track *rt_star_time;
const struct sync_track *rt_star_pers;
```

Tracknames should start with “rt_”. It is suggested that you name them according to their namespace and variable names, e.g. rt_part or rt_star_time.

7.3.2 sync_get_track()

Your internal sync_tracks are made known to the Rocket editor via the *sync_get_track* function. Its invocation is also the place to provide the names that appear in the editor and on disk for the saved tracks. In hellodemo's main.c this is all done in getSyncTracks():

```
void getSyncTracks() {
[..]
// stars
rt_star_time = sync_get_track(rocket, "stars(1):time");
rt_star_pers = sync_get_track(rocket, "stars(1):perspective");
[..]
}
```

Depending on the definition of SYNC_PLAYER, sync_get_tracks will either connect to the editor, or try to load the tracks from disk.

7.3.3 sync_get_val()

Getting variables out of Rocket is done via the *sync_get_val* function. It takes the desired trackname and row-position as input and returns a Float. The calculation of *row* is based on the play-time of the music. Implementation details can be found in drawDemo() and also in updateDemo().

The code snippet below shows how we get the variables for the stars effect:

```
void drawDemo(int time) {
[..]
int ri_star_time;
float rf_star_pers;
[..]
// stars
ri_star_time = (int) sync_get_val(rt_star_time, row);
rf_star_pers = (float) sync_get_val(rt_star_pers, row);
[..]
}
```

As per our convention, variables from Rocket that hold:

- *Float* values (the default) should be preceeded with *“rf_”*
- *Int* values (requiring an explicit cast) should be preceeded with *“ri_”*

7.3.4 Passing variables to an effect

Once we have the time-based values for our variables from Rocket, we can feed them to our effects. As written above, the effects to not know of Rocket. They just need to get their parameters when calling their render-function. All effects take the variable *time* as their first and most imporant parameter. They are required to render a frame according to *time*.

The stars effect additionally takes a perpective factor as a parameter.

```
void starsEffectRender(int time, float persfak);
```

7.4 Common Effect Functions

As previously implicitly covered (6.2.2), effects are expected to have typical functions when working our way.

More explicitly, the minimum required functions when coding single effects are:

```
void effectInit();
void effectRender(int time);
void effectRelease();
unsigned int* effectPalette();
```

Init, Render and Release should be clear. The Palette function is required, as the current palette will be used outside the scope of the effect, e.g. to support setting the brightness.

To avoid name-clashes, it is good practice to preceed those names with the actual effect name, e.g. starsEffectInit().

When putting the effects together as a demo, you need one additonal function:

```
void effectOn(int startTime);
```

This is a very short function that basically sets the screenmode, palette and dimensions (xres/yres) everytime an effect is called for the first time. This is required because you will want to have different effects that might run in different screen modes. Here is how the starsEffectOn() function looks like:

```
void starsEffectOn(int startTime) {
  starsEffect_startTime=startTime;
  xres=320;
  yres=180;
  wosSetMode(8, screenBuffer, starsPalette(), 0);
  g_currentPal = starsPalette();
}
```

The *startTime* is not always used, but always required by convention. *xres* and *yres* in this example are for a typical 16:9 low-res screen. *wosSetMode* sets up the display for that mode (see 10.2 for a list of available modes) to work from that frame-buffer *screenBuffer* and with the palette pointer obtained from *starsPalette()*. The initial brightness is set to 0 (black). Finally, the palette pointer is stored in the global variable *g_currentPal*.

7.5 Common Demo Functions

When putting your effects together as a demo, you would naturally call their *Init* and *Release* functions from the respective functions in the demo's main.c (here: *initDemo* and *deinitDemo*). Actually, in hellodemo, the init-functions are all grouped in another function called *reloadDemo*. This will become useful later when starting to use the file-watcher and auto-reload functionality.

7.5.1 drawDemo()

Then, when it is time to draw a frame, the demo needs to figure out which effect (part) needs to be called, and if it had already been shown in the previous frame. If not, the part has to be enabled. This is done like this:

```
int prevPart=-1;
[..]
void drawDemo(int time) {
  int curPart;
  [..]
  curPart = (int) sync_get_val(rt_part, row);
  // init part
  if ( (prevPart != curPart) | curPart==0) {
    switch (curPart) {
      case 0:
        pictureEffectOn(time); break;
      case 1:
      starsEffectOn(time); break;
      case 2:
      movetableEffectOn(time); break;
  default: break;
  };
  }
```

When it is verified that the part is enabled, it can be drawn like this:

```
[..]
// draw part
switch (curPart) {
  case 0:
  pictureEffectRender(ri_picture_id); break;
  case 1:
  starsEffectRender(ri_star_time, rf_star_pers); break;
  case 2:
  movetableEffectRender(ri_move_time, ri_move_xtab, \
  ri_move_ytab); break;
default: break;
};
```

Finally, at the end of the draw function, the prevPart variable it prepared for the next frame, and the current frame is displayed.

```
prevPart = curPart;
wosDisplay(2);
```

wosDisplay triggers the chunky to planar conversion, i.e. the conversion of the byte-chunked *screenBuffer* (of 8 bit colour depth, as set previously with wosSetMode) into 8 seperate bitplanes to be displayed by the Amiga AGA chipset. Due to the slow speed of the chip memory, the Amiga will spend almost an entire frame for this conversion in a typical low-res resolution. It is generally unrealistic to expect to be able to draw 50 frames per second (FPS). Depending on the effect complexity, the target hardware, and the frame's content, you would often be able to reach a framerate in the range of 25 to 40 frames per second. But a stable motion at a fixed framerate looks smoother than one with varying framerate. Therefore, wosDisplay takes a frame-lock parameter:

- 0 : no frame-lock, i.e. run as fast as possible (possible faster than 50 FPS in very low resolutions)
- 1 : frame-lock to every frame, i.e. 50 FPS max
- 2 : frame-lock to every other frame, i.e. 25 FPS max

While 50 FPS can be achieved with some 160x90 screenmodes, you would generally want to aim for 25 FPS with low-res 320x180 screenmodes.

7.5.2 updateDemo()

While most screenmodes will not update in every frame, there is a possibilty to run certain parts of the code at 50 FPS intervals by putting it in the provided *updateDemo* function. This is triggered by the vertical blank interrupt (VBI) from *assemly-bridge.s* and can be used for making quick changes, e.g. to the colour palette when fading.

Part III Annex

8 Credits

8.1 Framework

The Haujobb Amiga Framework has been developed by Hellfire and Noname.

- low-level hardware API *WickedOS* by Noname
- high-level PC API counterpart *WOS Qt* by Hellfire
- standard code in */shared* by Hellfire
- examples by Noname and Hellfire
- documentation by Noname

8.2 Third party code

Standing on the shoulders of giants. With little modifications here and there.

- system-friendly hardware startup by Piru
- system-friendly interrupt example by Comrade J
- Kickstart v39 sprites fix by Comrade J
- Speedychip by Piru
- C2P routines by Kalms
    - partially modified with inline-saturation code based on an idea from Blueberry by Hellfire and Noname
- Further WickedOS components (currently not exposed through API)
    - Tracker Packer 3 by Crazy Crack
    - The Player 6.1 by Guru
    - old AHX/THX by Dexter
    - AHX by Dexter
    - Profiler by Bartman
    - CRM decruncher by Thomas Schwarz
- ADPCM player in Hello Demo by Kalms (Amiga) and Ian Luck (PC)
- Rocket sync-tracker and library by Kusma et al.

8.3 Artwork

How would life be without artists? The following pieces have been bundled with the release of our framework:

- picture *Haujobb Forever* by Helge
- picture *Androids Dream* by Acryl
- movetable texture by JCS
- music in Hello Demo by Muffler

9 Trouble-Shooting

9.1 Missing Standard Headers

With Visual Studio Community 2017 we have witnessed missing standard headers, resulting in compilation problems even for the simplest programs such as the hello world (6.1).Unbelievable as it sounds, this behavior is also documented on Stack Overflow, along with the solution in answer #1.

To solve this problem, you need to rerun the Visual Studio Installer and select *Windows Universal CRT SDK* under *Individual Components -> Compiler, Buildtools and Runtimes.* Then restart you IDE, and rebuild your project.

10 WickedOS

This is a collection of notes about the WickedOS-layer of our Amiga Framework. Normally, you should not need to study this, but you are very welcome to.

10.1 Built-In Test

If you are feeling nostalgic or curious, you can assembly and run a built-in example of WickedOS in ASM-One or Devpac on the Amiga! The visual outcome will be the same as that described in section 2.3.Of course, it doesn't matter if you are using real or emulated hardware for this.

10.1.1 With ASM-One

If you coded on Amiga before, chances are that ASM-One is already installed and that you know how to use it. The minimum supported version is 1.29. If you do not have at least this one, just either:

- download ASM-One from http://aminet.net/package/dev/asm/ASM-One,
- or get ASM-Pro from http://aminet.net/package/dev/asm/AsmPro1.18src, which is an improved and open-sourced version
- or grab my copy of ASM-One v1.29.

Set the WOS: Assign before you start ASM-One.

1. assign “WOS:” to where you unpacked WickedOS

2. list WOS:wos_v#? (should list the main file, currently v1.62)
3. Consider putting the assign from line 1 into s:user-startup (optional)

Then test WickedOS' built-in example AGA-screen from within ASM-One.

1. Start ASM-One from a Shell (This is important, as starting it from an icon will cause you a lot of trouble, as previously discussed on Amiga Demoscene Archive.)
2. Give it some public memory
   1. ALLOCATE Fast/Chip/Publ/Abs>p
   2. WORKSPACE (Max. XXXXX) KB>1000
   3. Note: ASM-One from Aminet (v1.48) seems to expect an assign “Sources:” at startup. If you don't have it, either:
      1. assign “Sources:” to somewhere useful, or just to “Ram:” for testing
      2. or in Asm-One go to Preferences->Environment and change “Default Dir:” textbox value to an existing directory or empty it
      3. or restart Asm-One with “!” to get the memory prompts again.
3. Menu->Assembler->Preferences->Assembler
   1. CPU 68020
   2. Check “FPU Present”
   3. Uncheck “UCase = LCase”
   4. Uncheck “; Comment”
   5. Press “Save”
4. Type “v wos:” (to set the working dir)
   1. v (to list the dir, just to confirm and refresh your memory)
5. Type “r” (read) and select the latest version of wos:wos_v#?.s
6. Press <ESC> to get into edit-mode
   1. Remove the “;” before WTEST in line 3
   2. Press <ESC> again to go back into command line mode (“>”)
7. Type “a” (assemble), and ASM-One will assemble your source.
8. Type “j” (jump), and you should see a nice picture and listening to a tune. (If it didn't work, you might not have AGA, 68020+ and FPU in your machine. Please note that WickedOS does not strictly require neither AGA nor an FPU, but this configuration is the baseline for the AGA/060 demos that we want to build with it.)

10.1.2 With Devpac

Devpac was probably the best allround commercial assembler development package for the Amiga. If you want to give it a go, either use your existing installation or get a copy from the English Amiga Board file server (files “Devpac v3.04#?.adf” in /Commodore_Amiga/App/Disk/).

Once installed:

1. Project->Load the WickedOS main-file wos_v1.#?.s
2. Remove the “;” before WTEST in line 3
3. Settings->Assembler->Options
   1. Processor: 68020
   2. Check “68881/2 Maths Coprocessor”
4. Program->Assemble
5. Program->Run

Check the manual to learn about hotkeys, and the really powerful debugger “monam”. The manual is available at https://computerarchive.org/, which host lots of other Amiga manuals, too, in /files/comp/applications/amiga/manual.

10.1.3 Recap

You tested that the supplied WickedOS-sources directly assemble and work on the minimum supported assembler, ASM-One (without using intermediate object-files and linking). This provides the ground for the following steps with vasm and vlink.

If you were planning to make extensive use of WickedOS with ASM-One or Devpac on Amiga, you might be interested in setting the WOSASSIGN-flag. This would allow you to set your working directory to somewhere else and then just include WickedOS from wos:newestwos.s, which would then find all of its files from the WOS: assign.

But even if you don't plan to use ASM-One, you must at least provide the assign to INCLUDES: if you plan to assemble WickedOS on Amiga.

10.2 List of Screenmodes

todo

10.3 How to add a new screenmode

Adding a new C2P-based screenmode to WickedOS requires a few modifications at several places in several files. As I tend to forget at least one place for myself, this write-up is at least for my own reference.

Hint: you can set wtest flag in the main wos_v#?.s source for setup and display a simple screen. Note: XY is to be replaced with a running number, e.g. 19

in file wos#?.s

1. in _wosbase:
   1. mXY: dc.l 0,0,0,0 ; add some comments as well
2. in WOSInit:
   1. add an init block for your c2p-mode after the last one (.m18ok at the time of writing)

```
ifnd NOMODEXY
   lea mXYc2p,a0
   lea mXY,a1 ;dc.l 0,0,0
   move.l a0,(a1)+ ;Init
   add.l #4,a0
   move.l a0,(a1)+ ;Main
   add.l #4,a0
   move.l a0,(a1)+ ;Exit
endc .m7ok:; feel free to add custom init code here
```

```
; e.g. generation of ham masks or the like
```

3. in _Display:
    1. add macro-line "_DisplayM1 XY" in first block
    2. add macro-line "_DisplayM2 XY" in second block
4. in _RefreshDisplay:
    1. "add cmp.b #XY,d0" and "beq .mXY"
    2. provide fitting code at .mXY (copy from appropriate 8 bit or ham-based examples above)
5. in SetModeAndColors:
    1. add macro-line "_SetMaCM1 XY" in first block
    2. add macro-line "_SetMaCM2 XY" in second block
    3. (for ham-modes) add special cmp/beq cases at ".m0"
6. in _SetMode:
    1. add macro-line "_CallSetModeMac XY" in first block
    2. add macro-line "_SetModeMac XY" in second block
7. in _SetColors:
    1. add macro-line "_SetColM1 XY" in first block
    2. add macro-line "_SetColM2 XY" in second block
8. in Modes:
    1. provide fitting block which adds exactly one bit to MODES (double the number of the previous) e.g.
       ```
       ifnd NOMODE18
          MODES set MODES!131072
       endc
       ```
        1. (this bitwise-logic is why there are currently a maximum of 32 modes supported)
    2. include you binary code and provide label mXYc2p
        1. (code is expected to have jump table like all the other modes - init at 0, c2p at 4, etc..)

in file sub/wos_defines.i

1. in struct after "Define the structure" comment
    1. add four rs.l 1 entries to structure for modeXYinit, modeXYc2p, modeXYexit, modeXYptr
2. after "Set up the screen-dimensions" comment
    1. provide fitting MakeMode line to describe your screen dimensions
    2. (or alternative definitions of modeXYsize .. modeXYsize)

in file sub/wos_copperlists_v#?.s

1. at EOF
    1. provide fitting copperlist named CopmXY, also with labels sprmXY and colmXY with appropriate space

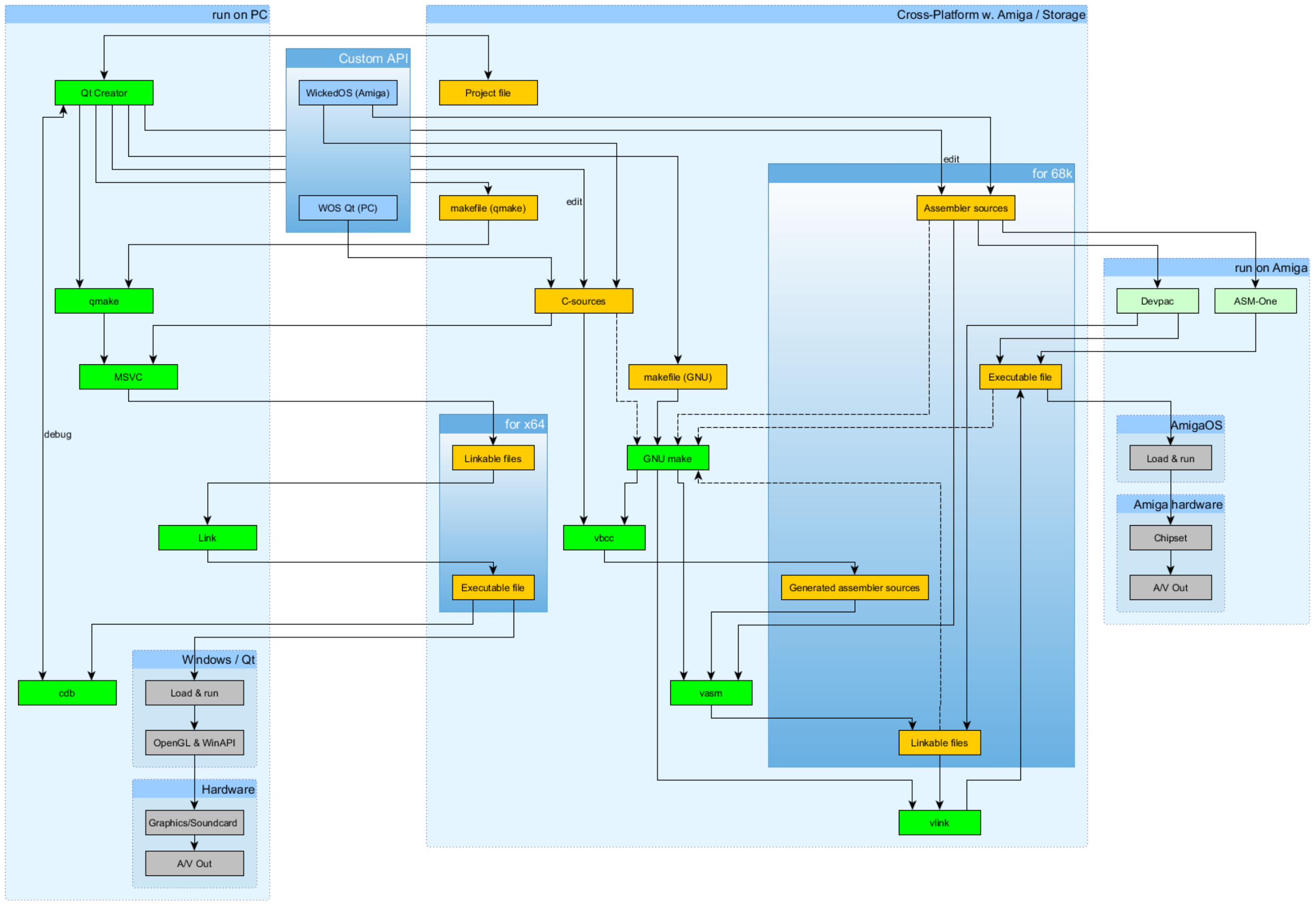
in dir sub/chunky/sources

1. Finally you need to provide a fitting c2p-routine as a PC-relative plugin. Check the provide sources and do likewise.
    1. name the file as "mode#?.s"

in file sub/chunky/sources/makefile

1. add filename to *Sources*

Schematic Overview



Todo:

- Advanced Framework Functions
  - profiler
  - file-watcher
  - cache simulator
  - palette tool
- run
  - visual studio command prompt
  - “nmake install” to get the required libraries to the folder
- mixasm
  - ace make all
- Audio, getting your music in
  - music dependencies
    - rocket_bpm
    - rocket_rpb
    - rocket_row_rate
  - stream player
    - bass on PC
    - Kalms ADPCM on Amiga
  - Audio ADPCM Saving and determining BPM
- Screenmodes
  - list of
  - example
- Literature
  - https://www.doc.ic.ac.uk/lab/cplus/cstyle.html