

This document contains three sections, a short history of this code, the latest porting details to Intel x86_64 architecture, and how I use Xcode to maintain everything.

History

I originally came across the FORTRAN code for Adventure and Zork around 1978. Although after 44 years the details are a bit fuzzy, it was probably from a Digital Equipment Corporation (DEC) machine of small-word size, at least when compared to the Control Data Corporation (CDC) machines I was using. Others ported Adventure initially to CDC land, while I did Zork. In those days ASCII was not common, indeed, most character sets were all upper case.

The first thing to note is that there was no character data type, so all vocabulary (Hollerith data), as well as octal bit-flags, were smushed together into large INTEGER arrays. Adventure's 4-character vocabulary fit nicely into a single INTEGER, but Zork's 6 character limit required two INTEGERS at the time. At least on DEC systems ... although RADIX-50 may have been used, I cannot recall.

My first Zork port was to CDC 170 hardware with 60-bit words and ten 6-bit Display Code characters per word, running the NOS operating system. So I ripped apart all the double-integer code and made it work with single-word INTEGERS. Of course, the driving reason was to reduce the array sizes and make the program smaller, which was an important factor when memory was measured in a handful of mega-words. The random text database file was generated with custom COMPASS assembler code that wrote a multi-record file, which was addressed randomly using a record number, I believe we called them indexed sequential files. I also added the Spider Room and an End Game, and when all was complete I renamed the program to Qork. Why? I guess because I knew the code would never see the light of day, and it was different than the original, and heck, I'd never even played the original Zork, Qork might be significantly different in ways I was not aware of!

Speaking of that, to test everything I had done I needed to actually play Qork, and the only way to do that was to have a map. So Carol and I made maps, using a state of the art flowchart template, a pencil and our hands! I've included what we did in 1980. The originals are somewhere yet to be determined, but I did find the high resolution PICT file scans, which I've cleaned up in Affinity Photo and converted to PNG. Keep in mind that the scans I made back then look great even today - Apple, always looking forward, must have used vector graphics. (Note: I have since discovered the original hand-drawn maps on line printer paper, yellowed with age!)

That was a lot of work, but I was young. I was able to play an entire game of Qork - with map in hand - in one hour!

Time passes ... it's now more than a decade later.

Around 1990 there was a bigger, better, faster computer in my life, a CDC 180 machine with 64-bit words and eight 8-bit ASCII characters per word, running the NOS/VE operating system. Like its 170 predecessor, this was big-endian architecture, we'll see the significance of that later. But now I have proper upper and lower case support, so, you guessed it, I converted every word, phrase and sentence of the Qork text database

from all upper to appropriately-cased English. I also moved from the custom assembler indexed sequential text database to a standard FORTRAN 77 direct access file - same idea as the indexed sequential file but now implemented in a portable manner. That was a lot of work, but I was young. I forget other details of the port, but other than SAVE/RESTORE there probably wasn't all that much involved, as everything was still using single INTEGER words for data structures.

This was when I took control of Adventure. It had been ported to NOS years previously, but to have it run on NOS/VE meant that I would have to do the work. As with Qork, the text database used a custom random access file format that I changed to standard direct access. Again, the biggest hassle was that the Adventure text database was not proper ASCII upper and lower case, it was Display Code upper and lower case, but Display Code is a 6-bit character set, and to get a lower case letter from a 6-bit upper case letter required an escape character, the caret '^'. So "Hello World!" in ASCII was "H^E^L^L^O W^O^R^L^D!" in Display Code. Yuck, that was a lot of work, but I was young.

Thus, in 1990 I had both Adventure and Qork running on NOS/VE, available to me and me only, basically. They were packaged in a state of the art Source Code Utility (SCU) library with automated System Control Language (SCL) build and run procedures. Other than those two games I have never played any games in my life, and I stopped playing them at that time. And matters sat that way for a quarter of a century.

It's now 2015 and I am old and starting to cleanup. Throughout my life I have managed to bring forward lots of stuff, from source code to media, for no real reason that I can discern, other than to have it around. There's not much reason to do that anymore, but for old times sake I decided to modernize my Adventure and Qork and release them into the wild of the public domain. What you will see here is FORTRAN code that has survived 4.4/10 of a century of time. I've left most of the old code vestiges intact, just commented out. The result is a messy looking source, but that is my intent, you are an implementer and are used to that sort of thing. The mere user still only sees pristine game play.

x86_64 Porting Details

It's taken the world FOREVER to move to 64 bit words. For me to port Qork to Linux required making the vocabulary and data INTEGER*8 types, that was the only way to get 6 characters per word. I was NOT going to backtrack to two integers per Qork word! And FORTRAN 77 these days is not FORTRAN 77 from 44 years ago, in particular gfortran is not happy with all the trickery using shifts, masks and the bitwise AND and OR operations used by the game input parsers to pack multiple A1 characters into an A8 INTEGER, left justified and space-filled. pgf77 and ifort could be coerced into that; however, x86_64 hardware is little-endian so I had to reverse the packing and the shifting, etc, it was a mess and for sure a PITA. To make things as portable as possible, I redesigned everything using the CHARACTER data type as an intermediary, and this is what I came up with, much nicer, hardware independent, and works on all the compilers available to me:

- The parser still accepts A1 input characters stored in INTEGERS (80a1), and returns packed A8 characters in INTEGER*8 words, for backwards compatibility.

- It converts 80a1 input data to CHARACTER*80 variable IN0:

```
write( in0, '80a1' ) (inbuf(i), i=1,80)
```
- Since this is all ASCII data, I used this code to single-case everything in IN0 to a new character*80 variable IN:

```
do i = 1, len(in0)  

  if ( ! defined __GFORTRAN__ )  

    j = ichar(in0(i:i))  

    if (j>= ichar("A") .and. j<=ichar("Z") ) then  

      in(i:i) = char(ichar(in0(i:i))+32)  

    else  

      in(i:i) = in0(i:i)  

    end if  

  #else  

    j = iachar(in0(i:i))  

    if (j>= iachar("A") .and. j<=iachar("Z") ) then  

      in(i:i) = achar(iachar(in0(i:i))+32)  

    else  

      in(i:i) = in0(i:i)  

    end if  

  #endif  

end do
```
- It was then a simple matter to loop through the CHARACTER*80 variable IN, collect non-space characters into the character variable WORD, and then encode them into the INTEGER*8 array OUTBUF:

```
read( word(1:8), 'a8' ) outbuf(outbufc)
```

But gfortran wasn't through with me yet, as it wasn't as flexible as pgf77 and ifort when it came to coercing character data to integer data. In fact, it plain wouldn't do it!. So all the millions of constructs like:

```
integer*8 vocab(689)
data vocab/ "LANTER", "TROLL", ....
```

I had to convert to:

```
integer*8 vocab(689)
data vocab/ 6hLANTERN, 5hTROLL, ...
```

I look at that as a digression caused by gfortran; then again, pgf77 and ifort are commercial compilers and having gfortran compatibility seemed like a worthwhile compromise ... free is what folks want.

And of course there was all the normal port-stuff like random number seeding and generation that was different and varied among the compilers. And Adventure did not have a SAVE/RESTORE feature so I added that, and a front-end procedure to manage game save states, and to build the games, databases and save states. And how to handle EOF and whether to use stdin or /dev/tty, and replacing sense switches with environment variables. And still more case conversion, and removing carriage control characters ... you do know what a carriage control character is, don't you? And I fully implemented the Qork TIME command so you know how much time you've spent playing the game.

That was a lot of work! Did I mention that I'm not young? Oh, I also fixed Guardian mode; too bad I didn't do that much sooner, my job would have been easier.

```
GDN>HE
VALID COMMANDS ARE :
AA- ALTER ADVS
AC- ALTER CEVENT
AF- ALTER FINDEX
AH- ALTER HERE
AN- ALTER RVARs
AO- ALTER OBJCTS
AR- ALTER ROOMS
AV- ALTER VILLS
AX- ALTER EXITS
DA- DISPLAY ADVS
DC- DISPLAY CEVENT
DF- DISPLAY FINDEX
DH- DISPLAY HACKS
DL- DISPLAY LENGTHS
DM- DISPLAY RTEXT
DN- DISPLAY RVARs
DO- DISPLAY OBJCTS
DP- DISPLAY PARSEr
DR- DISPLAY ROOMS
DS- DISPLAY STATE
DT- DISPLAY TEXT
DV- DISPLAY VILLS
DX- DISPLAY EXITS
D2- DISPLAY ROOM2
EX- EXIT
HE- TYPE THIS LIST
NC- NO CYCLOPS
ND- NO DEATHS
NR- NO ROBBER
NT- NO TROLL
PD- PROGRAM DETAIL
RC- RESTORE CYCLOPS
RD- RESTORE DEATHS
RR- RESTORE ROBBER
RT- RESTORE TROLL
TK- TAKE
GDN>
```

In summary, you have FORTRAN 77 source for Adventure and Qork that compiles with pgf77 (PGI), ifort (Intel) and gfortran (GNU). Each program has its associated text database that's read once during initialization and is converted to a direct access file that's indexed by a record number. Also during initialization the block data common blocks that define a game's state are output as an unformatted file. The random text database and initial saved game state are what is required to play a game, and what you, Implementer, must create for your users. I wonder if endian-ness problems remain?

Tested on Linux using gfortran, pgf77 and ifort. Tested on macOS 10.10 (Yosemite) with gfortran.

In 2015 I used gfortran-5.0-bin.tar.gz from <http://sourceforge.net/projects/hpc>.

I also tested on macOS 10.6.8 (Snow Leopard), although the binary built on 10.10.3 failed with Illegal Instruction, which I was never able to build-around by disabling hardware features like sse4. So I dug-up Xcode 4 and used gfortran 4.7 to make

special static binaries just for 10.6 (Snow Leopard), and as it turns out for 10.7 (Lion) too.

The 2018 update used <https://sourceforge.net/projects/hpc/files/hpc/g95/gfortran-7.1-bin.tar.gz/download>.

The 2020 update used <https://sourceforge.net/projects/hpc/files/hpc/g95/gfortran-8.3-bin.tar.gz/download>. (I first tried to use gfortran 9.2, but it would not compile my codes, -lSystem was missing.)

From Terminal, install thusly:

```
sudo -s                                # enter admin password
cd /                                    # important
tar -xvpf ~/Downloads/gfortran-8.3-bin.tar # --> /usr/local/bin/gfortran
```

But wow, Gatekeeper was a PITA. Took 3-4 OK cycles just to get the compiler running, then 5-6 OK cycles to run the resultant binary! After that **platag** ran just fine.

It's now 2022 and I've upgraded to gfortran 11.2.0 from <https://sourceforge.net/projects/hpc/>, which required game source code changes, and I fixed the missing -lSystem. The compiler installs just like in 2020:

```
sudo -s                                # enter admin password
cd /                                    # important
tar -xvpf ~/Downloads/gfortran-11.2-bin.tar # --> /usr/local/bin/gfortran
```

Thus:

- in 2015 TAG-1.0 there were 2 static executables for each game.
- in 2018 TAG-1.1 there were 3 static executables for each game.
- in 2020 TAG-1.2 there were 4 static executables & 1 Linux binary for each game.
- in 2022 TAG-1.3 there are 5 macOS (+ 1 Linux) static executables for each game.

Static Executables	Compiler Version	OS version
(Adventure I Qork)-10.6.bin	gfortran 4.7	macOS 10.0 -10.7
(Adventure I Qork)-10.8.bin	gfortran 5.0	macOS 10.8 -10.12
(Adventure I Qork)-10.13.bin	gfortran 7.1	macOS 10.13 -10.14
(Adventure I Qork)-10.15.bin	gfortran 8.3	macOS 10.15 - 11.6
(Adventure I Qork)-glibc-2.27.bin	gfortran 7.5	Ubuntu Linux 5.3.0-42
(Adventure I Qork)-12.3.bin	gfortran 11.2	macOS 12.0 - ?

The tool that both users and Implementers use is called **play-text-adventure-games** (**platag**), and that Perl script will pick an executable based on the macOS system version. Try **platag -full-help** for complete help.

Sadly, much of the source has been truncated to 72 columns and thus many valuable comments have been lost. Back then interactive time sharing systems often used columns 73-80 for sequence numbers.

macOS Specifics:

- The simplest way of distributing TAG is a tarball, suitable for Linux and macOS. For macOS I spent extra effort to create a spiffy disk image. But regardless of the packaging scheme macOS now makes it difficult to actually play the games because of Gate Keeper and friends. Starting with the 1.2 release and macOS 10.15 Catalina the distribution is now signed with my Developer cert and notarized, see **makeMacDist**.
- In TAG-1.3 I cleaned up all the code signing and notarization issues and the distributed DMG completely passes Gatekeeper muster, although this required moving away from simple command scripts: I had to make actual Adventure and Qork applications. So Linux users will, for example, run `Qork.command` while macOS users will click on `Qork.app`. The apps are simple shell script wrappers, but it took a long time before I found the proper bundle structure to please codesign.
- Perl: scripting languages have been deprecated, so I've included perl5.30.0 in the distribution. I built a static Perl with a minimal module library, no man pages, etc, which bypasses all the issues of loading dynamic libraries with relative paths. I've saved the **config.sh** file so rebuilding should be easy:
 - `curl -O http://www.cpan.org/src/perl-5.30.0.tar.gz`
 - `tar -xzf perl-5.30.0.tar.gz`
 - `cd perl-5.30.0`
 - (ensure **config.sh** is present in this directory)
 - `./Configure -S`. (NOTE: verify prefix =.)
 - `make`
 - `make install`

Xcode

After years of iOS development I found that I missed having Xcode around for this recent work, particularly for source code and asset management, revision history and examining diffs. What you see in the TAG distribution is only a subset of what I have, and I eventually realized that Xcode could manage all the extra stuff for me. Here are some examples of the files in my Xcode project:

- This PDF that you are reading is a Pages document.
- The original PICT map images.
- The Affinity Photo projects of the imported PICTs, with my enhancements.
- An OmniGraffle document of Adventure's All Alike Maze. The AAM was missing from my original Adventure map, so I overlaid a PNG image of the OmniGraffle maze as an Affinity Photo layer to create the final Adventure map that you find in the distribution.
- A shell script to create the distribution tarball. Really nothing special here, although because I do all my work on macOS, it builds special static versions of the Adventure and Qork executables so mere mortals can play without having to install gfortran. The gfortran incantation to do this differs from what you see in `play-text-adventure-games` by adding these command line arguments;
 - `static-libgfortran -lgfortran -lgcc -lSystem -nodefaultlibs /usr/local/lib/libquadmath.a`
- For the Linux static binaries I used Ubuntu and this invocation:

- ```
gfortran -static-libgfortran -lgfortran -lgcc -w -cpp -o Adventure-glibc-2.27.bin src/Adventure.f
```
- And, of course, all the TAG files that comprise the released distribution tarball.

The Xcode project is a simple External Build project, although I just use platag for my builds rather than a Makefile with multiple build targets. Still, the Xcode viewer and editor work well with the FORTRAN source code, and the built-in SCM features are convenient.

Have fun, Steve Lidie, [lusol@icloud.com](mailto:lusol@icloud.com).

2022.04.01