ABI - overview

Saturday, September 28, 2024 8:34 PM

Author: Damir Ljubic
Email: damirlj@yahoo.com

Application Binary Interface (ABI)

As the name itself implies, the ABI refers to the "*run-time interface*" that describes the interactions between either compiled code: binary and OS, and/or between the binaries themselves (like the apps and the shared libraries, or the apps communicate via external interfaces - IPC).
It specifies how the elements of the software: functions, data, system calls and library dependencies will be translated into machine language, so that binaries - even those compiled in different languages, can still communicate together.
This heavily depends on the targeting platform (run-time environment):

- its architecture (arm64_v8a, x86-64, MIPS, etc.),
- as well on the OS (Linux, Windows, iOS, etc.)

The key points are:

- **Calling convention**
  - How the data will be passed to functions
  - How the return value will be provided back to the caller
  - Who is responsible for cleaning the function stack (caller, or callee)
- **Data layouts**
  - How the data are stored into memory
  - The size and alignment of data in memory
  - Endianness (big vs. little endian)
- **System calls and dynamic libraries**
  - How the system calls are propagated (Linux: from caller in user-space, to callee in kernel-space, and returning the value - or storing the error indication)
  - How the dynamic library function are referenced from binary at run-time, using dynamic linker

Binary Interface Stability

Probably the biggest concern related with ABI is **backward compatibility** - ensuring that changes in your binary don't break the clients dependent code.

How to accomplish this?

- **Software design**

  - This can be part of the careful software design and fallowing the SOLID principles in order to reduce the propagating dependencies, like with Open-Closed Principle, to handle the new demands by extending the code, rather than to modify it.
  - For interfaces - especially for external interfaces as part of the IPC, one can use aggregated data types, rather than the APIs with multiple arguments, with reserving upfront some padding bytes at the end - for future extensions: so that at client side, the APIs, as well the memory footprints of the data (data layouts), remain the same.
    @Not recommending - there is a better way (continue reading).

- **Symbol versioning**

  Don't change the exported functions, or those exposed by the external interfaces, rather
  - Add new functions
  - In some languages, like Java - annotate the old functions as *deprecated*. This will signal the clients code that the new releases may stop maintain and remove these APIs.
  - In C++, you may try to use the *inline namespaces* for versioning

```
// Previous definition
namespace v1 { struct MyData {...}; }
```

```
// New definition - modified, or just extended
namespace inline v2 { struct MyData {...}; }
```

This will preserve the old definition within the previous namespace, while the new definition will be at the client side visible as *MyData* - without need to explicitly specify the full qualifiers (v2::MyData). At the "library" side - you can always specify the alias on the latest version

```
using my_data_t = v2::MyData;
```

and using it across the code.

In case that the *MyData* is part of the exposed functionality - the dependent implementation also need to be versioned.
@Guess what?
In C++ - starting with C++14, we can also mark the API deprecated, similar as with Java

```
// Backward compatibility
[[deprecated("@note This will not be supported in 2.3.4 release!")]]
namespace v1 { void foo(const MyData& data);}
namespace inline v2 { void foo(const MyData& data);}
```

At the client side, one can utilize on ADL (*Argument Dependency Lookup*) - to resolve the overloading function calls for unqualified functions, based on the namespace of the calling arguments

```
const auto data = v1::MyData {};
f(data);
```

- **Semantic versioning**

It's the way to communicate with client code release versions, indicated whether the ABI is broken - requires the client attention and code modification, or the changes are backward compatible.
The convention for that is
<major>.<minor>.<patch>, where

- major version, if changed - indicates the broken ABI,
- minor - some internal refactoring, adding the new features or deprecated the old one in a backward-compatible way, while
- patch is indication of some bug-fixes, or small changes in backward-compatible way.

For C++ code, one can specify the semantic versioning in cmake script

```
set_target_property(my_library PROPERTIES
    VERSION 1.2.0 // full semantic versioning
    SOVERSION 1 // major version - symbolic link
)
```

This will effectively create the symbolic links to the real version of the library that will be part of the binary LD_LIBRARY_PATH

We can specify the installation rules as well

```
install(TARGETS my_library
                LIBRARY DESTINATION <destination_path>)
```

On the target - you will have - at the destination path, something like

```
// Symbolic links
 my_library.so -> my_library.so.1
 my_library.so.1 -> my_library.so.1.2.0
 // The library itself
 my_library.so.1.2.0
```

On Android - the libraries are usually published to the Maven (private) repository - and will be resolved by Gradle at compile time: for compile-time dependencies, based on the specified semantic version.
There are plenty of other package managers - on different platforms.

Conan is one example of the package manager used for the C++ libraries

```
conan install <package-name>
```

My favorite is the APT (*Advanced Package Tool*) for Debian-based Linux distributions (for those who work on Ubuntu VMs)

```
// Install the package
sudo apt-get install <package-name>
// Install the package of a specific version
sudo apt-get install <package-name>=<version>
// Update packages indexes and upgrade all installed packages to the latest version
sudo apt-get update && apt-get upgrade
```