UDC 004.4:004.738.5(045)

O.Marchenko,
(National Aviation University, Ukraine)
B.Melnik
(National Aviation University, Ukraine)

Transfer methods of mobile applications from iOS to Android platform

Porting, portability and porting problem

Porting – in programming is an adaptation of some program or it part in order to make it work in other environment (platform), different from the environment under which it was originally written. This adaptation aims to save maximum of original programs custom properties.

This is the main difference between the concepts of “port” and “fork” – in the first case, all the custom features in the application are trying to save, while the second – is based on a common basis of independent development of new useful properties.

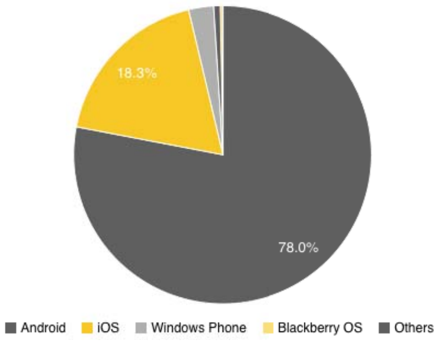
Porting process is also called “migrating” process, and the result of this process – “port”. But in any case, the main task of porting is to keep functionality, user interface and usual methods of working with the package and its properties. Adding new or removal of part of existing properties when porting software products is not allowed.

Portability usually refers to one of two things:

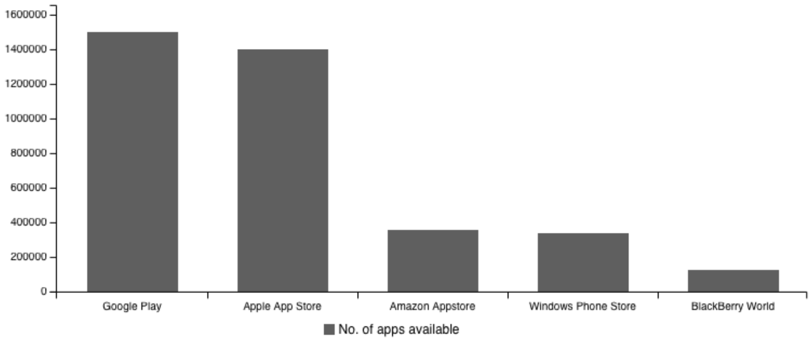
- Portability – as an opportunity to once compiled code, and then run it on multiple platforms without any changes.
- Portability – as a software feature that describes how easily the software can be ported. With the development of operating systems, programming languages and techniques, it is becoming easier to port software between different platforms. For example, one of the original goals of the C language and the standard library of the language -– has the ability to easily porting programs between incompatible hardware platforms. Additional advantages in terms of portability can have programs that meet specific standards and writing rules.

The need for implementation of porting usually arises due to differences in system command processor, the differences between the ways of interaction of the operating system and programs (API), the fundamental differences in the architecture of computer systems, or because of some incompatibilities, or even the complete absence of the programming language in the target environment.

Porting problem arises when there are two (or more) dominant platforms on the market, which takes most of market share (for example, like iOS an Android on mobile operating systems market), and when the customer has a desire to move his existing project from one to another platform, in order to increase the audience of his product. Best practice consists in using the parts of porting technique already on the early stages of application development.



Picture 1 – iOS and Android Market share diagram (Android – 78%, iOS – 18,3%) (Actually for 2016 year)



Picture 2 – Google Play Market and Apple App Store applications number diagram. (Google Play offers 1.5 million apps, and the Apple App store offers 1.4 million apps) (Actually for 2016 year)

Pictures 1 and 2 confirms existing of porting problem, in direction from iOS to Android as well as in direction from Android to iOS. (According to statistics, porting application in direction from iOS to Android is more widespread.)

Cross-platform code and main methods of writing cross-platform code

Source code can be considered as cross-platform, in case when this code transporting cost from one system to another is much less than the cost of writing new code from scratch. That is, we cannot call "porting" situation like this: "I have some kind of algorithm, I want to do the same algorithm on iOS - accordingly, I have to write the same algorithm for this platform. This is not the porting – this is writing code again. To port the algorithm, it should meet the cross-platform environment.

Main methods of writing cross-platform code can be defined as "Separation and Unification":

- First of all, a hard typification – it is important to write code in its common types, since different platforms built native-types can vary.

- Separation of code on algorithmic and non-algorithmic part. From algorithms pulls out the system calls and basic operations.
- Hardware-dependent part: we separate a program on the algorithmic part(math), and on the part that depends on the system.

Hard typification – is just defining of your own types for much better control, this is very helpful when you work, for example, with network, you can be sure that all of yours packages are int16, the distance between them is 0, there is concrete offset and there is no other way. After all, if you define all through a char, which suddenly becomes a double-byte, all go somewhere not there. Just to simplify the reading and writing code dependent on the platform / compiler / version of the language is very useful to introduce their own unified defines.

Hard typification sample (on C language – stdint):

```
#if defined(_MSC_VER)
typedef signed char         int8_t;
typedef signed short        int16_t;
typedef signed int          int32_t;
typedef signed __int64      int64_t;
typedef unsigned char       uint8_t;
typedef unsigned short      uint16_t;
typedef unsigned int        uint32_t;
typedef unsigned __int64    uint64_t;
#elif defined(LINUX_ARM)
typedef signed char         int8_t;
typedef signed short        int16_t;
typedef signed int          int32_t;
typedef signed long long    int64_t;
typedef unsigned char       uint8_t;
typedef unsigned short      uint16_t;
typedef unsigned int        uint32_t;
typedef unsigned long long  uint64_t;
#else
#include <stdint.h>
#endif
```

Algorithms and algorithmic part. For simplicity of porting, it's desirable that algorithm does not use any system functions, and even better we need to manage without any system libraries.

System calls. One of the basic principles of writing portable and managed code - do not use “malloc” (memory allocation) inside algorithms. Your algorithm should determine how much memory it required, and pass this value to your before written memory manager, which already allocates memory and provides a link to a dedicated part of initialization. The algorithm uses this part. This is right approach.

Basic operations. Under the basic operations involves the selection of non-specific operations that are not in C, but these operations are using very frequently. Example of basic operation can be CLZ (Count Leading Zeros) operation. To demonstrate the expediency of using basic operations, I’ll shoe simple example: a shift operation for int64. In principle, this command is in C, but may be implemented in various ways, since it is not standard int, this is int64. Int64 in different systems can have different names, but it's not so bad. The biggest problem is that this operation can be performed in different ways.

Hardware-dependent parts of mobile applications offer the following features:

- They heavily dependent on the API, frameworks, be it Android or iOS (or something else).
- They often use non-native tools. For example, code that uses video in Android can be such an algorithm: engine written in C, it calls the Java through JNI, which is then in turn pays back through JNI result.
- They use less portable languages. Less portable language I refer, for example, Objective-C and Java. Despite the fact that Java is considered to be cross-platform language, compared to C language with the universality it is clearly lacking.

Basic wrappers. In order to somehow unify the work with the hardware-dependent parts of the code, a good practice is to allocate a set of basic wrappers. There is my list of basic wrappers:

- System log: iOS – NSLog; Android – Logcat.
- Memory manager – must to perform malloc/free operation; As an option memory manager can perform seek of memory leaks operation.
- Mutexes\atomic operations – also different for different platforms.
- Thread manager – superstructure on mutexes and basic API for creating multithreaded applications.
- System information – can be useful if you change something in the runtime. If your code is optimized for multiple processes, you can learn in the runtime, what kind of system, and connect the chosen one part.
- Tracing\logging – it is not just the system log; This is debugging errors, and good if it is the same for all platforms.
- File managing – includes input\output files, .psm dumps and so on. Very handy to have a output from a single interface.
- Profiling Tools. If you have a heavy code, not always with help of standard tools you can quickly figure out what and where lost.

Additional wrappers. Depending on what makes the device, you may need additional wrappers. Here is a brief list of hardware parts that make mandatory of wrappers using (they need to porting process went smoothly):

- Sockets\network
- Audio devices
- Video devices
- Input devices
- Output devices

Most important things, that need to be considered when cross-platform application is creating

- Memory of data\code
- Data align

- Optimization for concrete processors
- Floating point
- Integer division
- Multithread an Main Thread.

Conclusions

Porting of software – important process in modern software engineering at all and at different markets, for example, like mobile operating systems market (most actually for iOS and Android).

Following all of the above recommendations, we can obtain an algorithm that is written through the hard types that you transfer. This algorithm does not require any intervention at all. Theoretically, it should compile and work immediately. Better to have unit tests to this algorithm. If the algorithm is compiled and passed inspection, most likely, all will be well. The basic operation is also can be ported, but they also may not be.

Hardware parts require porting. The best practice is when hardware parts have most closely to system interface. They will be easier to check, and they quickly ported.

Understanding of processor architecture improves portability.

The key to successful porting - this is a good cross-platform code.

Simplicity of portability defines forethought of architecture. So if you are writing a new application, conceived it cross-platform originally. Even if you do not plan to maintain it, perhaps it would have to be proceed by someone else.

References

1. Mooney (1997) – "Bringing Portability to the Software Process". West Virginia University. Dept. of Statistics and Computer Science. Retrieved 2008-03-17.
2. Garen (2007) – "Software Portability: Weighing Options, Making Choices". The CPA Journal. 77 (11): 3.
3. Lehey (1995) – "Porting UNIX Software: From Download to Debug". Retrieved 2010-05-27.
4. Sean Liao (2014) – “Migrating to Android to iOS Developers”. Apress.
5. Sean Liao, Mark Punak, Anthony Nemec (2014) – “Migrating to Swift from Web Development”. Apress.
6. https://mobiforge.com/research-analysis/mobile-software-statistics-2015
7. http://www.mobilephonedevelopment.com/porting-ios-to-android
8. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1
9. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=9&qpcustomb=1