Porting: A Development Primer

Mindfire Solutions
www.mindfiresolutions.com

January 14, 2001

Abstract:

This paper discusses software-porting process in details highlighting the steps involved in the process. It also talks about significant issues faced in a typical porting process and suggests appropriate strategies that should be used to make the whole process simpler.

INTRODUCTION: PORTING VS. FRESH DEVELOPMENT 2

- WHAT IS SOFTWARE PORTING 3
- TYPES OF PORTING 3

DIAGRAM: STEPS IN A TYPICAL PORTING PROCESS 4

FACTORS/ISSUES INVOLVED IN EACH STEP 6

- ASSESSING MIGRATION FEASIBILITY 6
- UNDERSTANDING INITIAL APPLICATION 7
- DECIDING TARGET DEVELOPMENT TOOLS 7
- VALIDATING ORIGINAL DESIGN PLAN 8
- DECIDING ABOUT PORTING STRATEGIES 8
- HANDLING RESOURCES (UI/HELP FILES) 9
- BUILDING, DEBUGGING AND TESTING 9
- PACKAGING AND RELEASING 10

OTHER SPECIFIC ISSUES 10

- CROSS-PLATFORM CONFIGURATION MANAGEMENT 10
- MAKE FILE / IDE PROJECT FILE 10
- SOURCE CODE NOT AVAILABLE 10
- CROSS DEVELOPMENT ON MORE THAN ONE PLATFORM SIMULTANEOUSLY 11
- WHAT HAPPENS WHEN THE ORIGINAL TARGET IS MOVING 11

CONCLUSION 11

Introduction: Porting vs. Fresh Development

During my involvement with some of the previous porting projects, I have been asked a typical question quite a few times. "Porting an application should have been easier compared to developing them from scratch, right?" Not going into the details about how I used to respond to those, let's analyze the factors that made that person assume so.

The very mention of the term Porting implicitly spells into a project where:

- Specifications are clear
- Detailed prototype in form of the running application is available
- Architecture and design is already there
- Identifying the subsystems are easier, as one can actually see the application running
- Source code is there, what else one can ask for?

Well that is that; now let us take a look at the flip side of the coin.

When put through the above question, my fellow developers came up with some interesting points which might make us think otherwise of a porting project.

The overall conclusion was that a porting project is anything but easier than a development project because:

- Specifications are generally biased towards the original platform/framework, hence they might actually be tougher for the target platform
- The application may not be well designed and architectured at all.
- Original architecture and design may not suite the target platform
- Linking the functional subsystems with their corresponding source files is a difficult task
- Understanding somebody else's code is always difficult
- It is easier to feel lost while looking at all the source code at once.

Now, this seems to give exactly opposite impression of porting projects. But the interesting thing is that even people, who were giving the above reasons for porting being difficult, couldn't deny the points given in favor of porting being easier. What the typical response to *"Specifications are generally clear in case of a porting project"* was

"Yes I agree, but you know, specifications are generally biased towards the original platform making it actually tougher"

You state any of the above points to support that porting is easier and the response you will get is going to have a common starting pattern: *"Yes it is so, but…"*

The above observation makes one thing clear that everybody at least agrees that a porting project could actually be easier but for some issues. So, instead of going further with the debate that whether porting is easier than development or not, let us try to find out the answer of slightly modified question, *"How to make porting easier?"*

• *What is Software Porting*

But first, "*What is software porting?"* Software porting is the engineering process of transforming an existing application so that the resulting software will execute properly on a new platform. The process involves the careful analysis; build, debug and test of the existing software to make sure that it will run reliably on the target. Depending on the nature of the software and the source and destination platforms, this can be as simple as recompiling the code on a new system and verifying that everything works properly, or as complex as redesigning the whole application and rewriting large sections of the code to accommodate the new platform.

It is important to understand that porting of an application is different from developing it from scratch. Just the fact that the original running application is available with all its source code, automatically removes some of the technical steps involved in the process as compared with that of a fresh development. But at the same time, it also introduces many decisional factors in the process.

• *Types of Porting*

Porting can be of different types mainly:

- Operating Systems e.g. *Mac* to *Windows*
- OS version *Mac OS 8/9* to *Mac OS X*
- Database *FoxPro* to *Oracle*
- Language *Fortran* to *C/C++*
- Framework/Libraries *Borland* to *MFC*
- Technology *COM* to *CORBA*
- Development Tools *VC++* to *CodeWarrior*
- Web platform *IIS/ASP* to *WebObjects*

A typical porting process can be a combination of more than one of the above type. For example, Porting of a typical MFC application from Windows to Mac automatically translates into followings:

OS porting from *Windows* to *Mac*
Framework porting from *MFC* to say *PowerPlant*
Development Tool porting from *VC++* to *CodeWarrior*

Every s/w project, including that of porting/migration, is unique. Although the actual process of porting might very well differ for different types of port, there are still some common issues faced in all of them and adopting appropriate strategies might make the task smoother and easier.

Diagram: Steps In A Typical Porting Process

I **Assessing Migration Feasibility**

1. Understand objective
2. Broadly identify the areas of the Source Platform covered by the original application
3. Identify Target Platform's restrictions
4. Make a Not Feasible/Not Possible list

Looks Feasible

No → Try to modify the original specifications to avoid the restrictions of target platform, keeping the functionality and behavior of application same

Yes

II **Understanding Initial Application**

1. Read user documentation to know the s/w functionality
2. Study original design plan, if available
3. Identify all hardware connections, Resource dependability and s/w interoperability
4. Visualize the source code (simple diagram could be helpful)
5. Go through the code, understanding the relationship between different functional subsystems and their corresponding source files

III **Deciding Target Development Tools**

1. List all the development tools used for the original application at source platform
2. Investigate about the availability of the same tools on target platform
3. Evaluate available Porting Tool options (e.g. conversion libraries, emulators)
4. Select the tools to be used at target platform based on the above findings

IV **Validating Original Design Plan**

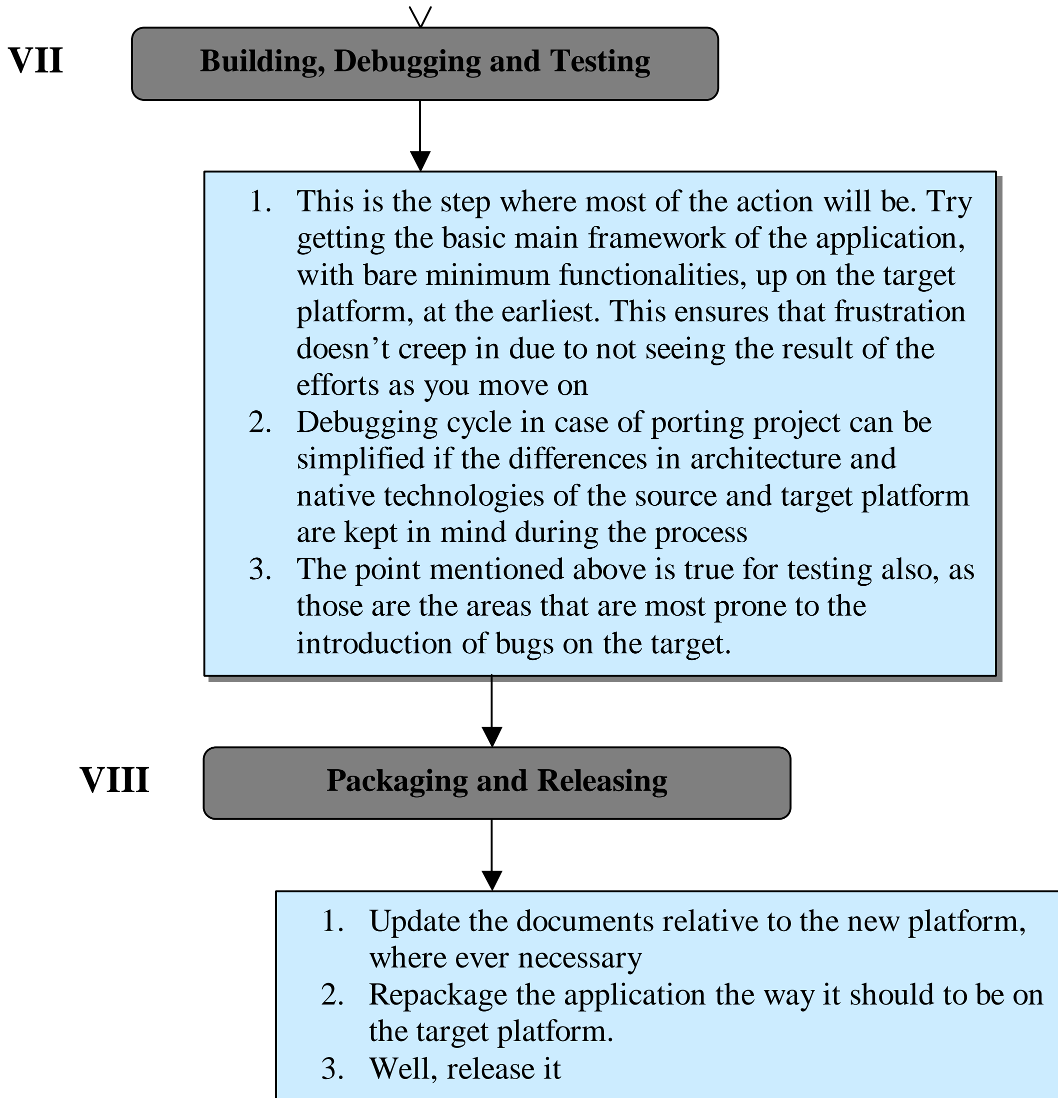
1. Strongly consider maintaining same code base
2. Identify the common code that can go unchanged across the target.
3. Identify the different layers in the application, and try to introduce wrappers as abstraction layer.
4. Re-validate the architecture and design of the source code and modify it, if required
5. Some of the known bugs on the source platform might be relatively easier to fix on target platform, Consider attacking them.

V **Deciding About Porting Strategies**

1. Prepare the launching platform by getting the dirty work (resources, help files etc.) out of the way early
2. Get the *Wrappers* ready both for code and data-types
3. Choose the coding strategy to be used, ++ or –
4. If feasible, try to get the code in buildable state early in the cycle
5. Do not deter from using target-specific technology, when it makes sense
6. Plan in advance, about how you are going to handle the issues present due to the architectural differences in source and target platform

VI **Handling Resources (UI/Help Files)**

1. Check if the current User Interface is appropriate for the new platform, change accordingly, if not
2. Decide upon the help format used on the target platform (you might want to use target platform specific format or generic cross platform format)



Factors/Issues Involved In Each Step

Although the steps mentioned above are self-explanatory, some issues need extra discussion to facilitate better understanding.

- ***Assessing Migration Feasibility***

Understanding the goal of the port is important. You should be very clear about what from the following the port is aiming at:

1. Getting a basic functional application running on the target platform. This is generally done as a marketing strategy to make the presence on the target platform felt at the earliest and is often followed by the full functional version within a suitable time frame.
2. Porting the existing application in its full strength on the target platform.

3. Porting the existing application adding new features and enhancements on the target platform. The fact that, for porting, the design and architecture of the original application is going to be visited again is often looked as an opportunity to introduce some additional features in the original. This essentially is like a new version release on the target platform.

Whatever the goal from the above, the project has, the trick (like any other s/w project) is to stick to the specifications and requirements once it is decided.

Next comes the feasibility study. Just because of the fact that original application is already running and running well on the source platform, making assumption that it can be ported to the target platform as it is, is a big mistake. Remember that more often than not, while deciding upon the specifications of the application, only the source platform was in consideration, so *specs* are bound to be inclined towards that side. Which means that on the target platform many of the specifications may not be feasible, or even, not relevant at all. Plus the target platform may have its own restrictions coming into your way. At the same time, the target platform will have its own strengths and special features, which might be easier and logical to exploit.

So, if you feel that the original specification should be altered, go ahead and make it a point. Anyway, the target user expects behavior consistent to his or her platform, so blindly mimicking is out of question.

- ***Understanding Initial Application***

You are not expected to, and you simply cannot, jump-start without acquiring necessary understanding of the original application. This includes reading user documentation to know about the s/w functionality and behavior in details. If the original design plan is available, going through that first will be helpful. To get the feel of the original source code, use some re-engineering tools or at the minimum, prepare a simple file/class/module level diagram.

Like for C++, making a comprehensive chart listing every interface files along with the headers included in that file, will give an idea about the inter-dependability of the files.

- ***Deciding Target Development Tools***

Deciding upon the suitable target development tools can be a critical step for simplifying the task of porting. If tools used on the source platform are also available for the target platform, they should be considered first. Though in some cases even if the tool used on the source platform is available on the target, it itself may not be matured or strong enough when compared with the other tools available on the target for the same purpose. In that case spending a little time on porting to the new tool/environment can actually save a lot of time in the long run.

Use of specific conversions or porting tools/libraries/emulators can also be of great help.

• *Validating Original Design Plan*

Effective porting is not just about making software for the target platform, it is also about making sure that ongoing releases on both the platform are easier. To achieve this the original design plan should be critically analyzed keeping the new targets in mind. This is also a good time to remove any existing flaw found in the original design (many a times, a design has to live with some known flaws, just because of the fact that they surface at a stage from where going back and removing them is not feasible in term of time/cost)

The other important aspect is to try to keep the code-base same on both the platform. Based on our experience, a "single source code-base" methodology is almost always the best. In other words, the version of the source code that is built for one platform is the same source code that is used to build it on another platform. By writing portable code (like, using conditional compilation where ever necessary), this is often a very achievable goal that will make long-term maintenance and enhancement of the product much more practical. For this identifying the portion of the code that can go unchanged on the target platform and dividing the whole code base into common/platform specific code will save a lot of effort later.

A properly designed application is likely to be divided into different layers maintaining a clean interface between them. In case of a typical desktop application, the pure application specific processing (engine) code can be in one layer while the GUI handling can be in different layer with a third abstraction layer sitting in between the two providing the interface to both. Maintaining this abstraction layers significantly reduces the amount of change required on the target platform.

The original application might contain some known bugs, which have not been fixed either due to not being technically feasible on the source platform or due to their close-to-release discovery. Consider attacking them along with the current port, since the issues that were not technically feasible on the source platform might be relatively easier to fix on target platform.

• *Deciding About Porting Strategies*

The strategies followed for a porting project might very well be project specific, but still there are certain thumb rules, which we can safely be applied to all of them.

Getting the code in buildable state early on the target platform should be given preference. It might mean initially adopting -- (minus-minus) strategy, i.e. take the whole project on the target platform and go on stripping (commenting/deleting) parts of the code till you get a buildable code (++ approach takes a new project and you keep adding functionality as the code moves). Then you can modify the part of the code commented in the previous step getting the functionality up one by one (following the ++ strategy).

Many a times it happens that even after putting a lot of effort in porting the code, you are not able to see the result just because the resources and other auxiliary things are not

ready. This is a dangerous situation and can lead to frustration. So, special attention should be given to getting the dirty work (resources, help files etc.) out of the way early.

If the design is having the abstraction layer then attack it first by getting the wrappers ready. Wrappers can be implemented for both code and data-types. Even if the original source code is not implementing the wrappers and instead using the platform specific data-structures and APIs, it is a good thing to implement them for the target platform. Doing so ensures maximum portability of the code and helps in maintaining the much desired *"single code base"*

The source and target platform may be quite different in architecture and technologies used. These differences might be due to the hardware used (different word size and byte order), due to the OS running (their different handling of inter-process communication, shared memory or security) or simply due to the differences in the tools used (different compilers/IDEs). The way these issues are going to be handled on the target platform should be planned in advance and the planned solution should be followed consistently over the whole project.

- ***Handling resources (UI/Help files)***

The current User Interface of the original application may be well designed for the source platform, but the target platform may have its own interface style, which might be quite different in look and feel from the source platform. Modify the current user interface appropriately to suite the target platform. At the same time, overall functionality and behavior of the application should remain consistent across the platform.

Original help files might be in a source specific format. While moving it to the target platform, it can either be converted to the target specific format or some generic cross platform format (like HTML). But in the later case, it is advisable to adopt the same format for the source platform also for its ongoing releases.

- ***Building, Debugging and Testing***

After walking through the above steps, the next is to actually make changes to the code so as to get them working on the target platform. If the factors as discussed in above sections have already been thought of carefully, this step can be relatively easier. Ignore the above steps and start porting the code right away, and you are guaranteed to be in trouble midway into the project.

Debugging cycle for a porting project is more or less similar to that of a development project except for the fact that it can be made easier if certain aspects are kept in mind. Majority of the bugs introduced will be due to the differences in technologies used for source and target. Like, if the application does file load/save and the byte order on the source and target is different, then that area is a potential source of bug introduction. If the above factor is considered while coding, a lot of *would be bugs* can be avoided.

Similarly, if the testers know that these are the areas where chances of bug introduction are high, they can plan their test activity accordingly.

- ***Packaging and Releasing***

The original documentations accompanying the source/product may not be in sync with that of ported one, so they should be reviewed and updated wherever necessary.

The final ported application should then be packaged according to the target platform. If maintaining the same source code base, then the new source should be integrated back to the main build tree for archiving and revision control. Even while maintaining different code base for target platform, the source patches for the bug fixes/enhancements in the ported application should be merged with the original source tree, so that the next release at the source platform is in sync with the ported application.

Other Specific Issues

Apart from the factors mentioned above there are other specific issues, which might have to be considered in some cases

- ***Cross-platform configuration management***

As mentioned above in *"Deciding Target Development Tools"*, you should try to use cross-platform tools that are available there on both the source and target platform. Among them, of utmost importance is, tool used for configuration management. Without a cross-platform source code configuration system in place, maintenance of the source could well turn out to be a nightmare, if one is trying to keep single source code base (which you should!).

- ***Make file / IDE project file***

Most often you will have access to project file for the IDE used on the source platform. Occasionally, you will just get the make file (or something similar) with the source code (well, you are never sure, be prepared for getting nothing at all). If you have the project file on the source side, then it is always nice to keep the same logical structure of the project on the target platform. In case you have the make file, use the "*import make file"* option that most of the latest IDE usually provide, to generate the project file.

- ***Source code not available***

Software projects can be as weird as you can think of. You can very well face a situation where you have to port the application from one platform to another, but you don't have access to the original source code, or the code is not useful. This might happen due to number of reasons:

1. The original source code is not in a maintainable state, so they want to start fresh.
2. The original application was developed and being maintained by some third party vendor, who is no more available.
3. The original application was developed using some XYZ language, which has been dead for long.

In any of the above situation, the best way is to look at the project as a fresh from-scratch development project with the extra benefit of having well defined specifications and a highly detailed prototype, in form of running original application, available.

- ***Cross development on more than one platform simultaneously***

In theory it is not a porting project, but thinking in a way that if it would have been a porting project then what type of source code you would glad to have so that porting is easier, should help in writing cross platform portable code (Abstraction layers, Wrappers, Common code base etc.) The benefit here is that you do not have to confide to the given architecture/design as in the typical porting project, rather you can explore more design options so as to suite both the platform.

- ***What happens when the original target is moving***

This is an interesting situation. You are on to porting an application that is yet not stable and is constantly moving. Well, if possible, then be in touch with the team working on the original application design and development and get the feel of the direction in which the application is moving, in advance. It is also advisable to plan your design meetings together, since there is no point in opting for a particular design on the source platform that is definitely going to create problems for the target platform.

Conclusion

It is important to realize that, fundamentally, software porting is as much an engineering process as any other software development effort. Understanding the goals, the overall process and the possible pitfalls will help you in taking well-informed decisions and adopting suitable strategies to counter the typical porting issues. By paying attention to the above principles, you can increase your chances for success and then if someone again remarks that the porting projects are easier, you can safely say: *"I can't generalize this, but yeah, I at least know how to make a porting simpler!"*

Mindfire Solutions is an offshore software services company in India. Mindfire possesses expertise in multiple platforms, and has built a strong track record of delivery. Mindfire passionately believes in the power of porting and its many advantages for software product companies.

We have developed specialized techniques to make porting efficient and smooth, and to solve the issues specific to porting. We offer core development and QA/testing services for your porting requirements, as well as complete life-cycle support for porting.

If you want to explore the potential of porting, please drop us an email at info@mindfiresolutions.com. We will be glad to help you.

To know more about Mindfire Solutions, please visit us on www.mindfiresolutions.com

---