# ECONOMIC AND TECHNOLOGICAL EFFICIENCY ISSUES IN THE SOFTWARE DEVELOPMENT PROCESS FOR THE OPERATION OF PROGRAMMING DEVICES

**Margarita YEGHIAZARYAN**
PhD in Economics, Department Chair, European University in Armenia
**Hayk GEVORGYAN**
NPUA, MA at Institute of TTEISSP
**Vigen KHACHATRYAN**
NPUA, MA at Institute of TTEISSP

Key words**:** economic efficiency, programmer, FPGA, programming, Windows, Linux, Java Native Interface, encapsulation, cross-platform

***Introduction***

This article examines the economic efficiency of developing methods and techniques for software delivery, focusing on processes managed by developers in Windows and Linux environments. Each method is described in detail, considering the specific intricacies of each operating system. Beyond the technological perspective, the article also addresses economic efficiency by optimizing programming processes to save resources, reduce development costs, and enhance workforce utilization. The proposed solution for Windows OS can be implemented in any programming language that can access the operating system's software interface, such as C/C++ or Delphi. The resulting solutions can be used with Java Native Interface technologies to create cross-platform Java toolkits, which help reduce development and operational costs.

***Methodology***

Programmable devices such as ROM chips, FPGAs, and microcontrollers are widely used today. These devices are commonly referred to as programmers. A programmer is a combination of hardware and software designed for writing and reading information in one-time programmable permanent memory devices [Kernighan & Ritchie, 1988, 272].

Typically, a programmer is an external device connected to a computer via a USB interface. Once connected, the device is recognized as a USB mass storage device and can be accessed as a file for reading and writing operations. The primary challenge lies in the development of an automated work environment (AWE) for the calibration and testing of quartz generators. The process of controlling and monitoring quartz generators is carried out using a programmer and an external frequency counter.

The calibration is performed as follows: coefficients and other parameters, such as amplitude values, divider frequencies, and input buffer types, are recorded into the device’s operational memory via the programmer. Then, using thermal cameras, the temperature is altered, and temperature-frequency characteristics are measured.

If the obtained results meet the required standards, the selected coefficients are permanently stored in the device's memory using the programmer.

From an *economic perspective*, the implementation of an automated work environment (AWE) can significantly reduce production costs by decreasing enterprises' dependence on manual calibration and testing processes. AWE not only reduces the impact of human factors but also accelerates production cycles, allowing for efficient resource allocation and a reduction in operational expenses. For such an AWE it is necessary to develop a programmer control module with the following requirements:

1. Cross-platform compatibility to ensure applicability across various environments.
2. No proprietary components from third-party vendors in the code to enhance cost efficiency and reduce dependencies.
3. Code reusability to minimize development and maintenance costs.
4. Since the core AWE code is implemented in Java, the module must be provided via Java Native Interface (JNI) to ensure flexible integration and reduce the need for additional development.

The article describes the module's implementation in Windows and Linux operating systems, considering both technological and economic factors that can contribute to improving the efficiency of software development.

***Literature review***

Software Development for the Programmer in Windows OS

To manage the programmer in Windows OS, it is possible to use the operating system's software interface, known as WinAPI. Interaction with the device requires the following functions [Hopcroft & Ullman, 1979, pp. 512]:

1. CreateFile – This function opens the device as a file for reading and writing. The device name is passed as a string: "\\.\PhysicalDrive t", where t represents the device number, starting from 1. It is important to distinguish between two versions of this function: CreateFileA and CreateFileW. The preprocessor selects one of these based on the encoding type. The first version supports Windows-1251, while the second supports UTF-8. If necessary, either can be explicitly used.
2. SetFilePointer – Sets the pointer for reading and writing data.
3. ReadFile – Reads a specified number of bytes into a buffer.
4. WriteFile – Writes a specified number of bytes to the device.

When using these functions, it is necessary to define certain parameters. To interact with the device, the following parameters are commonly used: "deviceName", GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0". These parameters allow obtaining the required level of access to the device.

After detecting the device, the pointer must be set for reading data using the **SetFilePointer** function. The programmer's device typically contains specific fixed-size sectors. To calculate the location of a specific sector, it is enough to pass the following parameters to the function: "hDevice, (SECTOR_NUMBER - 1) * SECTOR_SIZE, NULL, FILE_BEGIN", where *hDevice* is the device handle, and *SECTOR_NUMBER* and *SECTOR_SIZE* are the sector number and size, respectively.

After that, data reading can begin using the **ReadFile** function. It is necessary to first create two buffers: one for reading and one for writing. The call to the **ReadFile** function will look like this: "hDevice, bufferRead, SECTOR_SIZE, nb, NULL", where *bufferRead* is the buffer in which data of size *SECTOR_SIZE - 1* will be read, and *nb* is a variable where the number of bytes read will be recorded.

After reading the data, it is necessary to copy the contents of the read buffer into the write buffer. Before re-writing, the pointer should be repositioned again. To write, the pointer must be positioned at the beginning of the section where the data is to be written, and then the **WriteFile** function should be executed with the following parameters: "hDevice, bufferWrite, SECTOR_SIZE, nb, NULL". Data is written to specific sections of the device memory as defined by the programmer. Thus, interaction with the programmer can be carried out in Windows OS.

Below is an example code for connecting with the device in Windows:

```
bool connectToDevice()
{
   char deviceName[19] = "\\\\.\\PhysicalDrive ";
   for (int i = 1; i < 6; ++i) { deviceName[17] = i + 'O';
      hDevice = CreateFileA( deviceName,
            generic_read i generic_write, file_share_read i file_share_write, null,
            open_existing,
            file_attribute_normal,
            O
            );
      if (hDevice != INVALID_HANDLE_VALUE) {
         std::cout << "Some Device found!" << std::endl; SetFilePointer(hDevice, (SECTOR_NUMBER - 1) *
               SECTOR_SIZE, NULL, FILE_BEGIN);
```

```
        ReadFile(hDevice, bufferRead, SECTOR_SIZE, nb, NULL); for (int i = O; i < sizeof(bufferRead); ++i) { bufferWrite[i] = bufferRead[i];
        }
        return testSignature(bufferRead, signature);
      }
      std::cout << "Can't find device!" << std::endl;
   }
   return false;
}
```

*Software Development for the Programmer in Linux OS*

To develop software for programmer management in Linux OS, it is possible to use the operating system's software interface. The following functions are sufficient for interacting with the device [Knuth, 1997, pp. 800]:

- open – Establishes a connection with the device. It is defined in the fcntl.h header file.
- lseek – Sets the pointer for reading and writing data.
- read – Reads the specified number of bytes.
- write – Performs data writing.

As can be seen, these functions are very similar to the corresponding functions in WinAPI. For connecting to a block device, the following parameters are typically used: "deviceName, O_RDWR, O_SYNC, O_DIRECT". The **O_RDWR** flag allows for both reading and writing, while the **O_SYNC** and **O_DIRECT** flags ensure direct and synchronized access to the device.

As discussed in previous topics, before reading data, the pointer needs to be set. This is done using the **lseek** function with the following parameters: "hDevice, (SECTOR_NUMBER - 1) * SECTOR_SIZE, SEEK_SET", where **SEEK_SET** indicates the beginning of the file.

Reading and writing work the same way as in WinAPI, and they are defined in the **unistd.h** header file. This method will only work if the appropriate driver is installed in the system, allowing interaction with the device. If the driver is absent, direct communication with the device can be established through the UAS protocol [Tanenbaum, 2006, 1056], and the location where reading and writing need to be performed can be identified.

***Scientific novelty***

The article presents a new methodology for the combined application of programmer devices and software, which reduces the complexity of integrating hardware and software

components. The proposed solution is based on encapsulating low-level logic, allowing developers to interact with devices through high-level APIs without delving into technical details. This enables the creation of cross-platform solutions that ensure the efficient development of software systems across different operating environments (Windows, Linux). From an *economic perspective*, this approach significantly reduces software development, maintenance, and testing costs due to its modular structure and code reusability. The developed technology allows avoiding redundant programming for different platforms, which reduces the time to market for software products and ensures a high return on investment (ROI). The application of this methodology can stimulate technological innovations in the software industry by increasing productivity and the availability of cost-effective solutions [Pressman, 2010, pp. 929].

***Analysis***

*Combining C++ and Java*. The porting of C or C++ code can be accomplished through an automated generator that automatically generates the porting wrapper based on a provided interface. It is also possible to manually write this wrapper using Java Native Interface (JNI) functions. Below is an example of creating a porting wrapper using the "Simplified Wrapper and Interface Generator" (SWIG) tool [MacKie-Mason & Varian, 1994, 75-96]. From an *economic perspective*, combining C++ and Java plays a crucial role in reducing software development costs. The high-level flexibility of Java and the performance advantages of C++ enable the creation of efficient and cross-platform solutions without the need to develop separate software for each platform. This reduces development and maintenance costs and accelerates the implementation of new features [Nordhaus, 2007, pp. 48].

Assuming there is an original file written in C, to connect it to Java, it is first necessary to create a file with the "i" extension, which includes the information required to make it accessible from Java, such as functions, classes, etc. Below is an example of an interface file for the programmer.

```
%module Programmator

%{

tinolude "programmator.h"

%}

struct Coeff {

int inf, lin, scale, cub, four, fifth, vc, so, ampl, foh, div, of, oo, offset;

};

char* test();
```

```
bool connectToDevice();

bool isConnected();

bool setUc(int uc);

bool powerOn();

bool powerOff();

const char* readBurnedValues();

const char* readCurrentValues();

int send(struct Coeff coeff);

int burn(struct Coeff coeff);

const char* readKs();
```

After creating the interface, it is necessary to generate the porting file. This can be done using the following command:

"swig -c++ -java -package mypackage.wrapper interface_file.i"

Here, **mypackage.wrapper** is the name of the package where the generated files will be placed in Java, and **interface_file.i** is the name of the interface file. It is important to pay special attention to the package structure, as after the generation process, these files cannot be moved to other packages.

The ***economic advantages*** of this method include:

- *Cost reduction:* The use of high-performance base code in C++ reduces optimization issues that arise when using Java alone.
- *Efficient use of workforce:* Teams can work in parallel, with one focusing on the interface and business logic in Java, while the other focuses on low-level performance optimizations in C++.
- *Cross-platform development:* It allows avoiding redundant programming for different operating systems, reducing software maintenance costs [Dutta & Lanvin, 2023, 464].

As a result of executing this command, a porting file in C++ and corresponding source files in Java will be created. Before compiling them, it is necessary to include the Java Development Kit header files, specifically from the /jdk/include and /jdk/include/operative_system directories, where operative_system is the name of the operating system under which the porting is performed.

After completing all these steps, the C++ files will be compiled into a dynamic library. Before starting the porting tasks, it is necessary to place the dynamic library in the java.library.path or the system's path environment variable.

Finally, in the Java code, you will need to call System.loadLibrary("libraryName"), where libraryName is the name of your dynamic library without extensions.

***Conclusions***

As a result, a description of the interaction with developers has been implemented, where the device is represented as a USB Hash. Through the encapsulation of low-level logic, the ability has been created to abstract hardware characteristics at the software level. Thus, code written in C or C++ can be used in Java in an object-oriented manner, allowing the developer to work with the generated Java code without worrying about the low-level complexities of the USB device.

From an *economic perspective*, this approach offers several key advantages:

• *Cost reduction* – The encapsulation of low-level logic means that developers can work with higher-level languages, avoiding the need for complex low-level programming. This reduces the need for specialist retraining and lowers the software development budget.

• *Faster development* – The use of generated code in Java allows developers to work with intuitive APIs, saving time that would otherwise be spent on low-level engineering solutions.

• *Increased reliability* – Automated generation reduces the likelihood of errors, which in turn decreases the costs of future corrections and technical maintenance.

• *Reusability* – The resulting solution can be generalized and reused for working with different devices, saving on development and optimizing resource distribution.

The results obtained can be applied to interact with any device that supports such functions. The designed module can be generalized and used as a software library for high-level programming languages, such as Java. This will allow the development of similar programs using only high-level languages, which will accelerate development and reduce the potential number of errors.

***References***

1. Kernighan B. W., Ritchie D. M., *The C Programming Language*, Second Edition, Prentice Hall, USA, 1988, P. 272.
2. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, USA, 1979, P. 512.
3. D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Third Edition, Addison-Wesley, USA, 1997, P. 800.
4. A. S. Tanenbaum, *Operating Systems: Design and Implementation*, Third Edition, Prentice Hall, USA, 2006, P. 1056.
5. Pressman R. S., *Software Engineering: A Practitioner's Approach*, 7th Edition, McGraw-Hill, USA, 2010, P. 929.
6. Sommerville I., *Software Engineering*, 10th Edition, Pearson, USA, 2015, P. 816.

7. Brynjolfsson E., McAfee A., *The Second Machine Age: Work, Progress, and Prosperity in a Time of Brilliant Technologies*, W. W. Norton & Company, USA, 2014, P. 320.
8. Nordhaus W. D., *The Economic Impacts of Technological Advances: Innovation, Productivity, and Growth*, National Bureau of Economic Research, USA, 2007, P. 48.
9. Shapiro C., Varian H. R., *Information Rules: A Strategic Guide to the Network Economy*, Harvard Business Review Press, USA, 1998, P. 352.
10. Dutta S., Geiger T., Lanvin B. (Eds.), *The Global Innovation Index 2023: Innovation in the Face of Uncertainty*, World Intellectual Property Organization (WIPO), Switzerland, 2023, P. 464.
11. MacKie-Mason J. K., Varian H. R., *Economic Aspects of the Internet*, in *Economic Perspectives on the Internet*, Journal of Economic Perspectives, Vol. 8, No. 3, 1994, P. 75-96.
12. Aghion P., Howitt P., *Endogenous Growth Theory*, MIT Press, USA, 1998, P. 694.
13. Zhu K., Kraemer K. L., *Post-Adoption Variations in Usage and Value of E-Business by Organizations: Cross-Country Evidence from the Retail Industry*, Information Systems Research, Vol. 16, No. 1, 2005, P. 61-84.

**Margarita YEGHIAZARYAN, Vigen KHACHATRYAN, Hayk GEVORGYAN**
**Economic and technological efficiency issues in the software development process for working with programmable devices**
*Key words: economic efficiency, programmer, FPGA, programming, Windows, Linux, Java Native Interface, encapsulation, cross-platform*

The article discusses the key issues related to the economic and technological efficiency of software development involving programmable devices. It highlights the fundamental factors that influence the effectiveness of programming platforms and tools, considering both technological advancements and economic implications. As software development becomes more complex, optimizing resources and minimizing costs while maintaining high performance has become increasingly important.
The developed model provides software developers with a structured approach to reducing development time, lowering costs, and improving overall system reliability. One of the critical aspects covered in this study is the use of the Java Native Interface (JNI), which facilitates seamless integration of C++-written code within Java-based applications. This approach allows developers to leverage the advantages of both languages without requiring in-depth knowledge of low-level hardware interactions, simplifying the development process.
From an economic standpoint, this methodology significantly reduces initial investment costs and enhances the competitiveness of software solutions in the market. Furthermore, by streamlining development workflows, companies can lower operational expenses while ensuring better system performance. The proposed solution is highly adaptable and can be applied across various industries, including industrial automation, electronic device management, embedded systems, and high-performance computing. It opens new possibilities for software engineers to create scalable, efficient, and cost-effective solutions while maintaining flexibility across different technological platforms.