

KRUTSKIKH Elena Igorevna – Senior Lecturer of the Department of Software, Tver State Technical University, Tver. E-mail: yelena.korneeva@ya.ru

УДК 681.5.01

ОСНОВЫ ТЕХНОЛОГИИ ПЛАТФОРМЕННОГО ВЫЗОВА P/INVOKE В ПРОГРАММНОЙ ПЛАТФОРМЕ .NET И ЯЗЫКЕ ПРОГРАММИРОВАНИЯ C#

И.Т. Новиков, В.В. Лебедев

© Новиков И.Т., Лебедев В.В., 2026

***Аннотация.** В статье рассмотрены основные понятия и возможности технологии P/Invoke, предназначенной для вызова нативных (неуправляемых) функций внутри управляемой исполняющей среды CLR программной платформы .NET.*

***Ключевые слова:** C#, .NET, P/Invoke, неуправляемые функции, нативный код, маршалинг.*

Приложения на языке программирования C# разрабатываются на базе программной платформы .NET (бывшая .NET Core, предыдущая версия – .NET Framework). Платформа .NET приносит много удобств в программирование: сборщик мусора, JIT-компиляция, LINQ, большой функционал для языков программирования и многое другое. Однако наряду с преимуществами у .NET есть и минусы. Программы на платформе .NET компилируются в промежуточный язык программирования (CIL), который транслируется в машинный код с помощью виртуальной машины (CLR) [1]. Это сильно снижает производительность приложений, хотя и дает возможность реализовать весь перечень удобств, предоставляемый платформой. Тем не менее иногда программисту важна именно производительность, а в некоторых случаях у разработчика уже есть библиотека на быстром, «нативном» языке программирования, таком как C или C++. Возможно ли разработать приложение на языке C# в рамках платформы .NET с использованием уже ранее написанных на нативных языках библиотек, представленных машинным кодом, а не промежуточным языком CIL, который используется в .NET? На этот вопрос можно ответить положительно, так как платформа .NET предоставляет целый перечень возможностей для взаимодействия с нативными языками программирования. В этой статье мы рассмотрим такой механизм взаимодействия .NET и нативных языков, как P/Invoke.

P/Invoke (от «Platform Invocation Services» – службы платформенного вызова) – это специальный механизм, реализованный в .NET, который позволяет виртуальной машине CLR взаимодействовать с кодом, который написан на нативных языках программирования [2–6]. Благодаря этому приложения .NET могут вызывать функции из различного множества библиотек, написанных на нативных языках программирования, а также взаимодействовать с API операционной системы (ОС), так как ОС чаще всего написаны на C/C++.

В рамках P/Invoke код, написанный на платформе .NET (языки C#, F#, VB.NET), называют управляемым, так как за работу с памятью отвечает .NET (вернее, его сборщик мусора GC), а код, написанный на нативных языках (чаще всего C/C++), называют неуправляемым, так как CLR использует этот код, но не отвечает за то, как он работает с памятью ЭВМ [2, 4]. P/Invoke используется для вызова функций из библиотек, написанных на нативных языках программирования.

Для того чтобы язык программирования C# смог вызвать функцию нативной библиотеки, используют специальный атрибут – DllImport [3–6]. Рассмотрим простой случай использования DllImport.

Есть функция на языке программирования C или C++, которая находится в библиотеке MyLib.dll. Программист для вызова этой функции в приложении .NET должен знать ее сигнатуру: имя функции, возвращаемое значение и аргументы. Желательно знать и типы данных аргументов, особенно если они являются пользовательскими типами данных (структуры и классы). Функция также должна иметь модификатор «extern “C”», который служит указанием на то, чтобы компилятор обработал эту функцию, как будто она написана на языке C. В противном случае данная функция будет обработана в рамках языка C++ и компилятор добавит в файл библиотеки дополнительную информацию для нее, что затруднит взаимодействие с ней.

Для того чтобы C#-программист смог вызвать нативную функцию, он должен объявить ее прототип с модификатором «extern», чтобы указать компилятору на то, что у данной функции есть реализация, которая находится в другом месте кода, т. е. в нативной библиотеке. Функция должна также иметь модификатор «static». Это связано с тем, что C# – полностью объектно ориентированный язык, поэтому в нем нет полноценных функций, вместо которых используются статические функции класса.

Затем программист указывает для функции атрибут DllImport с параметрами. У атрибута есть много параметров, позволяющих настроить P/Invoke для этой функции. Значения аргументов атрибута DllImport приведены в табл. 1. Обязательным параметром для DllImport является путь к нативной библиотеке. Этот путь может быть полным, относительным, а кроме того, в качестве него может использоваться

просто имя библиотеки. В последнем случае CLR будет искать библиотеку в системных папках [4]. Примеры путей к системным папкам для разных ОС даны в табл. 2.

Таблица 1

Основные аргументы атрибута DllImport

Название аргумента	Дословный перевод на русский язык	Значение
dllName	Имя файла библиотеки *.dll	Путь к нативному файлу библиотеки. Может быть абсолютным или относительным. .NET сначала ищет файл в системных папках
EntryPoint	Точка входа	Указывает точное имя нативной функции. Можно использовать, чтобы обращаться к функциям со сложными именами более коротким и простым именем. Можно указать номер функции в *.dll-файле
CharSet	Набор символов (кодировка)	Определяет, как будут преобразовываться символьные литералы (тип char) и, как следствие, строковые литералы (string). Требуется в связи с тем, что в C# все строки используют кодировку Unicode, а в C/C++ символы (char) по умолчанию используют кодировку ANSI. Использование неверной кодировки может привести к ошибке вывода
SetLastError	Установить последнюю ошибку	Указывает, должна ли среда CLR запоминать ошибку, с которой завершилась нативная функция. Связано с особенностями обработки ошибок в языке C
CallingConvention	Согласование о вызовах	Определяет то, как будут передаваться аргументы из управляемой среды в неуправляемую, а также ответственного за очистку памяти

Окончание табл. 1

Название аргумента	Дословный перевод на русский язык	Значение
ExactSpelling	Точное написание	Указывает на то, что нужно искать нативную функцию с указанным названием. Этот параметр связан с тем, что в WIN32 API есть функции, которые работают с разными кодировками. Имена этих функций различаются специальным суффиксом
PreserveSig	Сохранить значение	Указывает, как обрабатывать дескриптор результата функции HRESULT: преобразовывать его в тип int или в тип исключения

Таблица 2

Каталоги, в которых DllImport ищет библиотеки

ОС	Каталог поиска файлов
Windows	Папка запуска приложения
	Текущая директория
	Системная папка (%WINDIR%/system32/)
	Системная папка для 16-битных систем (%WINDIR%/system)
	Папка «Windows»
	Каталоги из переменной среды
MacOS	Каталоги из переменной среды «DYLD_FRAMEWORK_PATH»
	Каталоги из переменной среды «DYLD_LIBRARY_PATH»
	Каталоги из переменной среды «DYLD_FALLBACK_FRAMEWORK_PATH» («~/Library/Frameworks/», «/Library/Frameworks/», «/Network/Library/Frameworks/», «/System/Library/Frameworks»)
	Каталоги из переменной среды «DYLD_FALLBACK_LIBRARY_PATH» («~/lib/», «/usr/local/lib/», «/lib/», «/usr/lib/»)

ОС	Каталог поиска файлов
GNU/Linux	Каталоги из переменной среды «LD_LIBRARY_PATH»
	Кешированный адрес библиотек из файла «/etc/ld.so.cache»
	Каталоги «/lib/» и «/usr/lib/»

У языка программирования C# есть важная особенность: в нем все является либо классом (String), либо структурой (Int32, Bool и т. п.). Из-за этого возникает проблема, связанная с тем, что управляемый и неуправляемый коды работают с совершенно разными данными. Для того чтобы преобразовать данные из управляемого кода в неуправляемый, используется технология маршалинга (Marshalling) [4–6].

При вызове неуправляемой функции платформа .NET преобразовывает (марширует) данные из управляемого кода (на языке C#) в неуправляемые данные языков C/C++. Когда нативная функция возвращает результат, .NET преобразовывает неуправляемые данные обратно в формат объектов .NET.

Пользователь может контролировать, когда нужно проводить процесс маршалинга: при переходе из управляемого кода в неуправляемый и наоборот. Для этого используются специальные атрибуты «In» и «Out», которые применяются к параметрам функций [2, 4]. Атрибут «In» указывает, что передаваемый управляемый параметр должен быть преобразован в неуправляемый тип данных, а атрибут «Out» – на необходимость обратного преобразования. Использование данных атрибутов позволяет повысить производительность, так как .NET не будет проводить маршалинг для этого аргумента при переходе его из одного кода в другой. Например, если программист хочет передать в функцию контейнер, в который должно записаться значение из неуправляемой функции, то смысла производить преобразование из управляемого типа данных в неуправляемый нет: контейнер по умолчанию пуст. В таком случае имеет смысл произвести преобразование из неуправляемого типа данных в управляемый, чтобы заполнить контейнер.

В C# некоторые типы данных являются blittable (неизменяемыми). Blittable-типы не требуют преобразования (маршалинга) при переходе из управляемого в неуправляемый код и наоборот. Обычно это простые числовые данные (int, double, byte и т. д.) [2, 4, 5].

В табл. 3 можно увидеть, какие типы данных относятся к изменяемым и неизменяемым при маршалинге.

Таблица 3

Деление типов данных в языке программирования C#
на blittable и non-blittable

№	Blittable-тип	Non-blittable-тип
1	Byte	Array
2	SByte	Boolean
3	Int16	Char
4	Int32	String
5	Int64	Class
6	UInt16	Object
7	UInt32	ValueType
8	UInt64	T[]
9	IntPtr	
10	UIntPtr	
11	Single	
12	Double	

Важным для P/Invoke является маршалинг non-blittable (неизменяемых) типов данных, к которым в первую очередь относят string, char, bool, все классы и структуры и многое другое [2, 4, 5]. Для работы с маршалингом можно использовать атрибут «MarshalAs», который относится к типу данных аргумента. Этот атрибут определяет, как будут преобразовываться типы данных аргумента при переходе от управляемого кода к неуправляемому. В качестве аргумента атрибут принимает перечисление.

Так, например, управляемый логический тип данных «Bool» в C# занимает в памяти 1 байт, в C++ – также 1 байт, а в чистом C типа данных bool нет: эквивалентом «false» в C является числовое значение «0», а «true» – все, что не есть «0». Из-за этого в Win32 API тип данных Bool занимает 4 байта: за ним скрывается тип int.

В более сложных сценариях можно реализовать собственный класс маршаллера для преобразования неуправляемых типов данных.

В случае если пользователь хочет получить или передать в неуправляемый код делегат (аналог указателя на функцию в C#), для этого используют атрибут «UnmanagedFunctionPointer». Так, если неуправляемая функция принимает в качестве аргумента указатель на функцию, то в управляемом коде функции можно будет передать делегат с этим атрибутом, который по своей сигнатуре совпадает с сигнатурой требуемого указателя на функцию [2].

Некоторые нативные функции могут принимать в качестве аргумента или возвращать из себя структуры или классы (в основном для C++). В C# тоже есть структуры и классы, но они имеют существенное

отличие от структур и классов в C/C++: среда CLR сама отвечает за расположение их полей. Из-за этого при простой передаче структуры/класса из управляемого кода в неуправляемый или наоборот может произойти ситуация, когда маршаллер запишет или считает не то поле, какое нужно: обычно для C/C++ необходимо, чтобы поля структуры/класса были расположены в том же порядке, что и в исходном коде.

Для того чтобы CLR не перемещал поля структур или классов в памяти, разработчик может использовать атрибут «StructLayout» со значением «LayoutKind.Sequential». Данное указание CLR на то, что поля структуры/класса должны быть расположены в памяти в порядке их объявления в исходном коде, часто применяют к структурам языка C#, так как он ближе всего по своему поведению к структурам языка C. Атрибут «StructLayout» также может иметь значение «LayoutKind.Explicit». В таком случае программист может вручную определить расположение полей структуры в памяти с помощью атрибута «FieldOffset», который указывает на смещение поля структуры относительно ее начала. Такой сценарий является более редким, но он позволяет реализовывать объединение (union) из языка C [2, 4].

Атрибут «StructLayout» также имеет один дополнительный атрибут для работы с выравниванием – Pack. Его использование связано с тем, что в программах C/C++ в структурах и классах компилятор может добавлять к полям пустые байты, чтобы поле имело адрес в памяти, кратный размеру типа этого поля. Разработчик может переопределить это поведение, чтобы экономичнее использовать память компьютера. Если разработчик на C/C++ переопределяет поведение выравнивания, то для корректного маршалинга структуры/класса необходимо указать правильное значение аргумента «Pack» в атрибуте «StructLayout».

В некоторых сценариях неуправляемые функции работают с различными указателями. Для того чтобы язык C# мог взаимодействовать с указателями, в нем предусмотрены структуры «IntPtr» и «UIntPtr», которые представляют собой целое знаковое и беззнаковое число соответственно [2]. Их размер зависит от архитектуры ЭВМ, так как на 32-битной машине длина указателя тоже будет 32 бита. Использовать обычные числа в качестве аналога указателей вместо этих структур не рекомендуется, так как это может привести к многочисленным ошибкам. Данные структуры имеют много сценариев применения: указатель на дескриптор, на функцию, на неуправляемые данные и т. п. Структура UIntPtr используется в основном для работы с арифметикой указателей, так как адреса памяти компьютера не имеют отрицательных значений. У статического класса «Marshal» есть множество статических методов для работы со структурами IntPtr и UIntPtr, в первую очередь для

преобразования их содержимого в тот управляемый тип данных платформы .NET, который подходит для дальнейшей работы.

Использование технологии P/Invoke в приложениях .NET имеет весомые достоинства:

- возможно использование множества нативных библиотек на языках C/C++;

- можно вызывать нативные функции в .NET-приложениях при критических требованиях к производительности, так как функции на C/C++ работают быстрее, чем на C#.

При этом использование P/Invoke имеет и ряд недостатков:

- приложение сложнее разрабатывать и тестировать, поскольку возникает необходимость в маршалинге, а отследить, что делает нативная функция из инструментов отладки .NET, невозможно;

- некоторые нативные функции могут передать в управляемый код неуправляемую память, которую программист должен будет очистить самостоятельно, без использования сборщика мусора;

- существует сложность с переносимостью приложений, так как приложение .NET должно ссылаться на нативные библиотеки, т. е. необходимо, чтобы .NET-приложение переносилось в связке с нативными библиотеками либо чтобы в приложении был механизм, самостоятельно определяющий расположение нативных библиотек;

- нативные библиотеки могут быть платформенно зависимыми и работать только на определенных ЭВМ (x86, x64, ARM64, RISC-V и т. д.), т. е. возникает проблема переносимости приложения на другие ЭВМ;

- при многократном вызове нативной функции происходит снижение производительности, поэтому желательно вызывать функции однократно в нужный момент.

Помимо технологии P/Invoke, в .NET существуют и другие способы взаимодействия с нативным кодом, такие как использование специального языка C++/CLI и функций статического класса NativeLibrary [2, 5]. Однако P/Invoke все еще остается одним из основных способов взаимодействия с неуправляемым кодом, так как он проще, гибче и универсальнее остальных.

В статье рассмотрен только базовый фундамент P/Invoke, так как эта технология обладает большим количеством настроек, что делает ее гибкой, но отразить все тонкости технологии в рамках одной статьи невозможно. Данный базис позволяет понять предназначение P/Invoke и особенности его использования в простых сценариях.

Таким образом, технология P/Invoke расширяет возможности программной платформы .NET и языка программирования C# и позволяет программисту использовать высокопроизводительные функции языков программирования C/C++, но при этом от разработчика требуется лучше продумать архитектуру и алгоритмы приложения, чтобы максимально воспользоваться всеми преимуществами P/Invoke.

Библиографический список

1. Попов Е. С# и .NET | Введение // Metanit.com. URL: <https://metanit.com/sharp/tutorial/1.1.php> (дата обращения: 14.10.2025).
2. Microsoft Learn: официальный сайт. URL: <https://learn.microsoft.com/ru-ru/> (дата обращения: 14.10.2025).
3. DllImportAttribute Класс. URL: <https://learn.microsoft.com/ru-ru/dotnet/api/system.runtime.interopservices.dllimportattribute?view=net-9.0> (дата обращения: 14.10.2025).
4. Interop with Native Libraries. URL: <https://www.mono-project.com/docs/advanced/pinvoke/> (дата обращения: 14.10.2025).
5. Интероперабельность с нативным кодом через платформу .NET. // habr.com. URL: <https://habr.com/ru/companies/otus/articles/809275/?ysclid=mgwid771t8264388768> (дата обращения: 14.10.2025).
6. Взаимодействие с DLL. URL: <https://www.rsdn.org/article/dotnet/coopdll.xml> (дата обращения: 14.10.2025).

FUNDAMENTALS OF P/INVOKE TECHNOLOGY IN A SOFTWARE PLATFORM .NET AND THE C# PROGRAMMING LANGUAGE

I.T. Novikov, V.V. Lebedev

Abstract. *The article discusses the basic concepts and capabilities of the P/Invoke technology, designed to invoke native (unmanaged) functions inside the managed CLR runtime of the .NET software platform.*

Keywords: *C#, .NET, P/Invoke, unmanaged functions, native code, marshalling.*

Об авторах:

НОВИКОВ Илья Тимурович – магистрант, ФГБОУ ВО «Тверской государственный технический университет», Тверь. E-mail: ilyanovickov@mail.ru

ЛЕБЕДЕВ Владимир Владимирович – кандидат технических наук, доцент, профессор кафедры электронных вычислительных машин, ФГБОУ ВО «Тверской государственный технический университет», Тверь. E-mail: Lebedev_vl69@mail.ru

About the authors:

NOVIKOV Ilya Timurovich – Master's Student, Tver State Technical University, Tver. E-mail: ilyanovickov@mail.ru

LEBEDEV Vladimir Vladimirovich – Candidate of Technical Sciences, Associate Professor, Professor of the Department of Electronic Computing Machines, Tver State Technical University, Tver. E-mail: Lebedev_vl69@mail.ru