УТВЕРЖДЕНО

МДЕА.50060-01 01-ЛУ

ПРОГРАММНЫЙ КОМПЛЕКС «МОНО»

Руководство оператора

МДЕА.50060-01 01

Листов 61

2022

Инв. № подл.		Подпись и дата		Взам. инв. №		Инв. № дубл.		Подпись и дата	

СОДЕРЖАНИЕ

1. НАЗНАЧЕНИЕ

Программный комплекс «Моно» предоставляет средства разработки и среду исполнения, совместимые с Microsoft .Net Framework, для операционной системы Linux. Программный комплекс создан на основе свободного ПО Mono и дополнительных средств, необходимых для разработки защищённых решений для российских операционных систем.

Необходимость чёткого разделения среды исполнения и среды разработки связана с требованиями эксплуатации защищённых систем, аттестованных по правилам ФСТЭК и Минобороны России.

1.1. Средства разработки

«Моно» предоставляет средства разработки для создания кросс-платформенных приложений. Приложения могут запускаться как в среде исполнения «Моно», так и других средах, совместимых с .Net Framework 4.7.

«Моно» включает все необходимые компоненты для сборки кросс-платформенных приложений:

— компилятор Roslyn C#;
— система сборки MSBuild;
— стандартная библиотека .Net Framework;
— ассемблер промежуточного кода;
— компилятор ресурсов;
— дизассемблер промежуточного кода;
— инструментарий сборки пакетов RPM и DEB.

1.2. Среда исполнения

Среда исполнения обеспечивает функционирование приложений, разработанных для среды .Net. Ядро среды исполнения сертифицировано по требованиям защиты и информации и включает:

— интерпретатор промежуточного кода;
— набор библиотек из состава стандартной библиотеки .Net Framework;
— библиотеку GDI+ для базовой функциональности WinForms;
— объектно-реляционную библиотеку Entity Framework;
— модуль Entity Framework для работы с СУБД PostgreSQL;
— математическая библиотека Math.NET.

Ядро среды исполнения «Моно» позволяет запускать приложения, разработанные по требованиям защиты информации ФСТЭК и Минобороны России.

Дополнительные инструменты среды исполнения:

— JIT-компилятор на базе LLVM;
— модуль Apache для запуска приложений ASP.NET;

— библиотека Gtk#.

2. СРЕДСТВА РАЗРАБОТКИ

2.1. Краткое введение

Инструментарий разработки «Моно» позволяет создавать приложения, работающие с конфиденциальной информацией и подготовленные для работы в среде приложения «Моно».

Приведем короткий пример приложения. Приложение проверяет наличие среды «Моно», открывает соединения с СУБД MSSQL и PostgreSQL, создаёт и заполняет БД.

Config.cs

```
using System;

namespace testMono
{
    public static class Config {
        public static string pgSqlConnectionString {
            get => "Host={адрес сервера};Database={имя БД};
            Username={имя пользователя};Password={пароль};";
        }

        public static string msSqlConnectionString {
            get => @"Data Source = {адрес сервера}\{экземпляр сервера};
            Initial Catalog = {имя БД}; Persist Security Info = True;
            User ID = {имя пользователя}; PWD = {пароль};";
        }
    }
}
```

Program.cs:

```
using System;

namespace testMono {
    class MainClass {
        public static void Main(string[] args) {
            try {
                Console.WriteLine(Type.GetType("Mono.Runtime").FullName);
            } catch (Exception e) {
                Console.WriteLine("Can't find Mono.Runtime type: " + e.Message);
            }
            Console.WriteLine("CLR version: ", Environment.Version.ToString());
            TestConnection.start();
            TestFeature.start();
            TestEntityFramework.start();
        }
    }
}
```

TestConnection.cs:

```
using System;
using System.Data.SqlClient;

namespace testMono {
    public class TestConnection {
        public static void start() {
```

```
            Console.WriteLine(nameof(TestConnection) + ": ", "start");
            try {
                checkMsSqlConnection();
                Console.WriteLine("MsSQL: ", "Ok");
            } catch (Exception e) {
                Console.WriteLine("MsSQL: ", "failed", e.Message);
            }

            try {
                checkPgSqlConnection();
                Console.WriteLine("PostgreSQL: ", "Ok");
            } catch (Exception e) {
                Console.WriteLine("PostgreSQL: ", "failed", e.StackTrace);
            }
            Console.WriteLine(nameof(TestConnection) + ": ", "finish");
        }

        static void checkMsSqlConnection() {
            var mssqlConnection = new SqlConnection(Config.msSqlConnectionString);
            mssqlConnection.Open();
            mssqlConnection.Dispose();
        }

        static void checkPgSqlConnection() {
            var npgsqlConnection
                = new Npgsql.NpgsqlConnection(Config.pgSqlConnectionString);
            npgsqlConnection.Open();
            npgsqlConnection.Dispose();
        }
    }
}
```

TestEntityFramework.cs

```
using System;
using System.ComponentModel.DataAnnotations.Schema;
using System.Data.Entity;
using System.Linq;

namespace testMono
{
    [Table("simple2column")]
    class SimpleData {
        public int Id { get; set; }

        public string data { get; set; }
    }

    class MainManageModel : DbContext {
        static MainManageModel() {
            Database.SetInitializer<MainManageModel>(null);
        }

        public MainManageModel() : base(Config.pgSqlConnectionString) {
        }

        public DbSet<SimpleData> Data { get; set; }
    }

    class TestEntityFramework {
        public static void start() {
            Console.WriteLine(nameof(TestEntityFramework) + ": ", "start");
```

```
            try {
                GetFromModel();
                Console.WriteLine("GetFromModel:", "Ok");
            } catch (Exception e) {
                Console.WriteLine("Entityfrmaework: ", "failed", e.StackTrace);
            }

            Console.WriteLine(nameof(TestEntityFramework) + ": ", "finish");
        }

        static void GetFromModel() {
            var context = new MainManageModel();
            var list = context.Data.ToList();
        }
    }
}
```

TestFeature.cs

```
namespace testMono {
    class TestFeature
    {
        public static void start() {
            Console.WriteLine(nameof(TestFeature) + ": ", "start");

            ValuTupleTest();

            Console.WriteLine(nameof(TestFeature) + ": ", "finish");
        }

        static void ValuTupleTest() {
            var t = ("lab50", 1.0);

            var (str, v) = t;

            if (str != "lab50" || v != 1.0)
                Console.WriteLine("ValueTuple: ", "failed", "");

            Console.WriteLine("ValuTuple:",  "Ok");
        }
    }
}
```

testMono.csproj

```
<?xml version="1.0" encoding="utf-8"?>
<Project Sdk="Microsoft.NET.Sdk" ToolsVersion="Current">
  <PropertyGroup>^M
    <OutputType>Exe</OutputType>
    <AssemblyName>testMono</AssemblyName>
    <AssemblyVersion>1.0.0.0</AssemblyVersion>
    <TargetFramework>net472</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System.ComponentModel.Composition" />
    <Reference Include="System.ComponentModel.DataAnnotations" />
    <Reference Include="System" />
    <Reference Include="System.Data" />
  </ItemGroup>

  <ItemGroup Condition="true">
```

```
    <Reference Include="EntityFramework"/>
    <Reference Include="EntityFramework.SqlServer"/>
    <Reference Include="EntityFramework6.Npgsql"/>
    <Reference Include="Npgsql"/>
  </ItemGroup>
</Project>
```

Для сборки и запуска тестового приложения необходимо выполнить следующие действия.

1) Заполните Config.cs актуальными данными подключения к СУБД.
2) Запустите команду `msbuild -t:restore`.
3) Соберите проект командой `msbuild -t:Rebuild -p:Configuration=Release`.
4) Запустите программу: `./bin/Release/net472/testMono.exe`.

Требуются следующие пакеты «Моно»:

— mono-devel;
— mono-microsoft-net-compilers-toolset;
— libentityframework6-npgsql-cil.

2.2. Параметры сборки

В этом разделе описаны параметры компилятора C#. Параметры сгруппированы в отдельные разделы по функциональному назначению.

В проектах .NET параметры компилятора можно задать двумя способами:

— *В файле *. csproj*

можно добавить свойства MSBuild для любого параметра компилятора в файле *. *csproj* в формате XML. Имя свойства совпадает с именем параметра компилятора. Значение свойства задает значение параметра компилятора. Например, в следующем фрагменте файла проекта задается свойство LangVersion.

```
<PropertyGroup><LangVersion>preview</LangVersion></PropertyGroup>
```

— *С помощью командной строки*

Каждый параметр компилятора можно использовать в двух формах записи: `-параметр` или `/параметр`.

2.2.1. Параметры языка

Следующие параметры определяют, как компилятор интерпретирует языковые функции. Новый синтаксис MSBuild выделен `полужирным шрифтом`. Для старого синтаксиса **csc** используется формат `code style`.

`CheckForOverflowUnderflow` / **`-checked`**

Создание проверок на переполнение.

`AllowUnsafeBlocks` / **`-unsafe`**

Разрешение использовать небезопасный код.

`DefineConstants` / **`-define`**

Определение символов условной компиляции.

`LangVersion` / `-langversion`

Указание версии языка, например, `>default` (последняя основная версия) или `latest` (последняя версия, включая дополнительные версии).

`Nullable` / `-nullable`

Включение контекста, допускающего значение `NULL`, или предупреждений, допускающих значение `NULL`.

2.2.1.1. CheckForOverflowUnderflow

Параметр `CheckForOverflowUnderflow` указывает, будет ли использование целочисленного арифметического оператора, в результате выполнения которого получено значение, выходящее за установленный для определенного типа данных диапазон значений, приводить к возникновению исключения во время выполнения.

```
<CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
```

Действие параметра `CheckForOverflowUnderflow` не распространяется на целочисленный арифметический оператор, находящийся в области действия ключевых слов `checked` или `unchecked`. Если в результате выполнения целочисленного арифметического оператора, находящегося за пределами области действия ключевых слов `checked` или `unchecked`, получено значение, выходящее за установленный для определенного типа данных диапазон значений, и для параметра `CheckForOverflowUnderflow` установлено значение `true`, во время выполнения возникнет исключение из-за этого оператора. Если параметр `CheckForOverflowUnderflow` имеет значение `false`, использование такого оператора не приводит к возникновению исключения во время выполнения. Значение этого параметра по умолчанию – `false`; проверка переполнения отключена.

2.2.1.2. AllowUnsafeBlocks

Параметр `AllowUnsafeBlocks` разрешает компилировать код, в котором используется ключевое слово `unsafe`. Значение по умолчанию для этого параметра – `false`, то есть использование небезопасного кода запрещено.

```
<AllowUnsafeBlocks>true </AllowUnsafeBlocks>
```

2.2.1.3. DefineConstants

Параметр `DefineConstants` определяет символы в файлах исходного кода вашей программы.

```
<DefineConstants>name;name2</DefineConstants>
```

Этот параметр задает имена одного или нескольких символов, которые необходимо определить. Параметр `DefineConstants` действует так же, как директива препроцессора `#define`, но применяется ко всем файлам проекта. Символ остается определенным в файле исходного кода до тех пор, пока определение не будет отменено с помощью директивы `#undef` в файле исходного кода.

При использовании параметра `-define` директива `#undef` в одном файле не действует для других файлов исходного кода в проекте. Вы можете использовать символы, созданные этим параметром, с директивами `#if, #else, #elif` и `#endif` для условной компиляции исходных файлов. Компилятор C# сам по себе не определяет символы и макросы, которые можно использовать в исходном коде. Все такие определения задаются пользователем.

Примечание. Директива C# #define не поддерживает присваивание значения символу, как, например, в языке C++. Например, с помощью директивы #define нельзя создать макрос или определить константу. Чтобы определить константу, используйте переменную enum. Для создания макросов в стиле C++ необходимо использовать альтернативные способы, например универсальные шаблоны. Поскольку макросы по своей природе подвержены ошибкам, в C# они запрещены, однако вместо них предлагаются более безопасные альтернативы.

2.2.1.4. LangVersion

Инструктирует компилятор принимать только синтаксис, включенный в заданную спецификацию языка C#.

```
<LangVersion>9.0</LangVersion>
```

Допустимы следующие значения.

Значение	Пояснения и комментарии
`preview`	Компилятор допускает использование любого допустимого синтаксиса языка из последней предварительной версии
`latest`	Компилятор принимает синтаксис из последней выпущенной версии компилятора (включая дополнительный номер версии)
`latestMajor (default)`	Компилятор принимает синтаксис из последней основной версии компилятора
`10.0`	Компилятор принимает только синтаксис, включенный в спецификацию C# 10 или более ранних версий
`9.0`	Компилятор принимает только синтаксис, включенный в спецификацию C# 9 или более ранних версий
`8.0`	Компилятор принимает только синтаксис, включенный в спецификацию C# 8 или более ранней версии
`7.3`	Компилятор принимает только синтаксис, включенный в спецификацию C# 7.3 или более ранней версии
`7.2`	Компилятор принимает только синтаксис, включенный в спецификацию C# 7.2 или более ранней версии
`7.1`	Компилятор принимает только синтаксис, включенный в спецификацию C# 7.1 или более ранних версий
`7`	Компилятор принимает только синтаксис, включенный в спецификацию C# 7.0 или более ранней версии
`6`	Компилятор принимает только синтаксис, включенный в спецификацию C# 6.0 или более ранней версии

Значение	Пояснения и комментарии
5	Компилятор принимает только синтаксис, включенный в спецификацию C# 5.0 или более ранней версии
4	Компилятор принимает только синтаксис, включенный в спецификацию C# 4.0 или более ранней версии
3	Компилятор принимает только синтаксис, включенный в спецификацию C# 3.0 или более ранней версии
ISO-2 (или 2)	Компилятор принимает только синтаксис, включенный в спецификацию C# ISO/IEC 23270:2006 C# (2.0)
SO-1 (или 1)	Компилятор принимает только синтаксис, включенный в спецификацию ISO/IEC 23270:2003 C# (1.0/1.2)

Версия языка по умолчанию зависит от целевой платформы приложения, а также от установленной версии пакета SDK.

Параметр компилятора LangVersion не влияет на метаданные, на которые ссылается ваше приложение C#.

Так как каждая версия компилятора C# содержит расширения для спецификации языка, параметр `LangVersion` не позволяет использовать возможности, аналогичные возможностям более ранней версии компилятора.

Кроме того, хотя обновления версий C# обычно совпадают с основными выпусками .NET, новый синтаксис и функции не обязательно привязаны к этой конкретной версии платформы. Для работы с новыми версиями требуется обновление компилятора, которое также выпускается с новой редакцией C#. Тем не менее для каждой функции определен собственный набор минимальных требований к API .NET или общеязыковой среде выполнения, в результате чего их можно выполнять на более ранних версиях платформы, добавив необходимые пакеты NuGet или другие библиотеки.

Независимо от того, какой параметр `LangVersion` вы задали, используйте для создания файлов с расширением .exe или .dll. текущую версию среды CLR. Единственным исключением являются дружественные сборки и ModuleAssemblyName, которые работают при установке параметра `-langversion:ISO-1`.

2.2.1.5. Nullable

Параметр `Nullable` позволяет указать контекст, допускающий значение `NULL`. Его можно задать в конфигурации проекта с помощью тега `Nullable`:

```
<Nullable>enable</Nullable>
```

Аргумент должен иметь одно из следующих значений: `enable`, `disable`, `warnings` или `annotations`.

Аргумент `enable` разрешает контекст, допускающий значение `NULL`. При указании `disable` контекст, допускающий значение `NULL`, будет отключен. Аргумент `warnings` включает контекст предупреждения, допускающий значение `NULL`. При указании аргумента annotations будет включен контекст заметок, допускающий значение `NULL`.

Примечание. Если значение не задано, применяется значение disable по умолчанию, однако .NET 6 шаблонов по умолчанию предоставляются со значением, допускающим enable NULL.

Анализ потока используется для определения допустимости значений `NULL` в переменных в исполняемом коде. Выводимая допустимость переменной значения `NULL` не зависит от объявленной в переменной допустимости значения `NULL`. Вызовы методов анализируются, даже если они условно опущены. Например, `Debug.Assert` в режиме выпуска.

Вызов методов, снабженных следующими атрибутами, также повлияет на анализ потока:

— Простые предварительные условия: `AllowNullAttribute` и `DisallowNullAttribute`
— Простые постусловия: `MaybeNullAttribute` и `NotNullAttribute`
— Условные постусловия: `MaybeNullWhenAttribute` и `NotNullWhenAttribute`
— `DoesNotReturnIfAttribute` (например, `DoesNotReturnIf(false)` для `Debug.Assert`) и `DoesNotReturnAttribute`
— `NotNullIfNotNullAttribute`
— Постусловия элемента: `MemberNotNullAttribute(String)` и `MemberNotNullAttribute(String[])`

Важно. Глобальный контекст, допускающий значения NULL, не применяется для созданных файлов кода. Независимо от этого параметра, контекст, допускающий значение NULL, отключен для любого исходного файла, помеченного как созданный. Существует четыре способа пометки файла как созданного:1)В файле editorconfig укажите generated_code = true в разделе, который применяется к этому файлу.2)Вставьте <auto-generated> или <auto-generated/> в комментарий в верхней части файла. Он может находиться в любой строке комментария, однако блок комментариев должен быть первым элементом в файле.3)Имя файла следует начинать с TemporaryGeneratedFile_4)В конце имени файла следует указать .designer.cs, .generated.cs, .g.cs или .g.i.cs.

2.2.2. Параметры вывода

В данном разделе описаны параметры компилятора C#, которые управляют выходом компилятора.

Следующие параметры управляют созданием выходных данных компилятора.

MSBuild	**csc**	Описание
`DocumentationFile`	`-doc:`	Создание XML-файла документации из комментариев `///`
`OutputAssembly`	`-out:`	Указание выходного файла сборки
`PlatformTarget`	`-platform:`	Указание разрядности ЦП целевой платформы
`ProduceReferenceAssembly`	`-refout:`	Создание базовой сборки
`TargetType`	`-target:`	Указание типа выходной сборки

2.2.2.1. DocumentationFile

Параметр `DocumentationFile` позволяет поместить комментарии из документации в XML-файл. Значение указывает путь к выходному XML-файлу. XML-файл содержит комментарии в файлах исходного кода компиляции.

```
<DocumentationFile>path/to/file.xml</DocumentationFile>
```

Файл исходного кода, содержащий метод `Main` или инструкции верхнего уровня, выводится в XML первым. Имя XML-файла должно совпадать с именем сборки. XML-файл должен находиться в том же каталоге, что и сборка. Если при компиляции не используется параметр `<TargetType:Module>`, `file` будет содержать теги `<assembly>` и `</assembly>`, которые указывают имя файла, содержащего манифест сборки для выходного файла.

Примечание. Параметр DocumentationFile применяется ко всем файлам в проекте. Чтобы отключить предупреждения, связанные с комментариями документации для определенного файла или раздела кода, используйте директиву #pragma warning.

Этот параметр можно использовать в любом проекте в стиле пакета SDK для .NET.

2.2.2.2. OutputAssembly

Параметр `OutputAssembly` позволяет задать имя выходного файла. Выходной путь указывает на каталог, куда помещаются выходные данные компилятора.

```
<OutputAssembly>folder</OutputAssembly>
```

Укажите полное имя и расширение файла, который требуется создать. Если не указать имя выходного файла, то используется имя проекта для указания имени выходной сборки. К проектам со старым стилем применяются следующие правила:

— EXE-файлу будет присвоено имя файла исходного кода, который содержит метод `Main` или инструкции верхнего уровня.

— DLL-файлы и NETMODULE-файлы берут имя из первого файла исходного кода.

Все модули, созданные в процессе компиляции, будут связаны со сборкой, полученной в результате этого процесса. С помощью `ildasm` можно просмотреть связанные файлы в манифесте сборки.

Параметр компилятора `OutputAssembly` обязателен, если требуется установить EXE-файл в качестве целевого для дружественной сборки.

2.2.2.3. PlatformTarget

Указывает, в какой версии среды CLR может запускаться сборка.

```
<PlatformTarget>anycpu</PlatformTarget>
```

— `anycpu` (по умолчанию) позволяет компилировать сборку для запуска на любой платформе. Если возможно, приложение выполняется как 64-разрядный процесс, а если доступен только 32-разрядный режим, переключается на него.

— `anycpu32bitpreferred` (по умолчанию) позволяет компилировать сборку для запуска на любой платформе. Приложение выполняется в 32-разрядном режиме в системах, поддерживающих и 64-разрядные, и 32-разрядные приложения. Можно задать этот параметр только для проектов, предназначенных для .NET Framework 4.5 и выше.

— `ARM` компилирует сборку для выполнения на компьютере с процессором Advanced RISC Machine (ARM).

— `ARM64` компилирует сборку для выполнения 64-разрядной средой CLR на компьютере с процессором Advanced RISC Machine (ARM) с поддержкой набора инструкций A64.

— `x64` компилирует сборку для запуска в 64-разрядной среде CLR на компьютере, поддерживающем набор инструкций AMD64 или EM64T.

— `x86` компилирует сборку для запуска в 32-разрядной среде CLR с архитектурой x86.

— `Itanium` компилирует сборку для запуска в 64-разрядной среде CLR на компьютере с процессором Itanium.

Параметр `anycpu32bitpreferred` является допустимым только для исполняемых файлов (.exe). Он используется только с .NET Framework 4.5 и выше.

В .NET Core, .NET 5 и более поздних выпусках параметр `anycpu` имеет некоторые особенности. При установке `anycpu` опубликуйте приложение и выполните его с помощью x86 или x64 `dotnet.exe`. Если вы используете автономные приложения, на этапе `dotnet publish` упаковывается исполняемый файл для настройки RID.

2.2.2.4. ProduceReferenceAssembly

Параметр `ProduceReferenceAssembly` указывает путь к файлу, в который нужно выводить базовую сборку. В API Emit он преобразуется в `metadataPeStream`. `filepath` указывает путь для базовой сборки. Обычно он совпадает с путем к файлу основной сборки. Согласно рекомендуемому соглашению (используемому в MSBuild), базовую сборку следует помещать во вложенную папку `ref/` относительно основной сборки.

```
<ProduceReferenceAssembly>filepath</ProduceReferenceAssembly>
```

Базовые сборки являются особым типом сборки, которая содержит только минимальный объем метаданных, необходимый для представления общедоступного API библиотеки. Такие сборки включают в себя объявления для всех элементов, которые важны при указании ссылки на сборку в средствах сборки. Базовые сборки исключают все реализации элементов, а также объявления закрытых элементов, не имеющих наблюдаемого влияния на их контракт API.

Параметры `ProduceReferenceAssembly` и `ProduceOnlyReferenceAssembly` являются взаимоисключающими.

2.2.2.5. TargetType

Параметр компилятора `TargetType` можно задать в одной из следующих форм:

— `library`: для создания библиотеки кода. `library` – это значение по умолчанию.

— `exe`: для создания EXE-файла.

— `module`: для создания модуля.

— `winexe`: для создания программы.

— `winmdobj`: для создания промежуточного файла *.winmdobj*.

— `appcontainerexe`: для создания EXE-файла для приложений Магазина Windows 8.x.

Примечание. Если вы создаете приложения для .NET Framework и не указываете значение module, этот параметр приводит к помещению манифеста сборки .NET Framework в выходной файл.

```
<TargetType>library</TargetType>
```

При каждой компиляции компилятор создает только один манифест сборки. В манифест сборки добавляются сведения обо всех файлах в компиляции. При создании нескольких выходных файлов в командной строке может быть создан только один манифест сборки, который добавляется в первый выходной файл, указанный в командной строке.

При создании сборки можно указать, что код полностью или частично CLS-совместим с атрибутом `CLSCompliantAttribute`.

2.2.2.6. library

Параметр `library` указывает компилятору создавать библиотеку динамической компоновки (DLL), а не исполняемый файл (EXE). Библиотека DLL создается с расширением *.dll*. Выходной файл получает имя первого входного файла, если только с помощью параметра `OutputAssembly` не указано иное. При построении файла *.dll* метод не требуется.

2.2.2.7. exe

Параметр `exe` указывает компилятору создать исполняемое (EXE) консольное приложение. Исполняемый файл создается с расширением EXE. Используйте параметр `winexe` для создания исполняемого файла программы. Если не указано иное с помощью параметра `OutputAssembly`, имя выходного файла совпадает с именем входного файла, который содержит точку входа (метод `Main` или инструкции верхнего уровня). В файлах исходного кода, который компилируется в EXE-файл, должна содержаться только одна точка входа. Если код содержит несколько классов с методом, указать, какой класс содержит метод, можно с помощью параметра компилятора `StartupObject`.

2.2.2.8. module

Этот параметр указывает компилятору не создавать манифест сборки. По умолчанию выходной файл, созданный при компиляции с этим параметром, имеет расширение *.netmodule*. Файл без манифеста сборки не может быть загружен в среду выполнения .NET. Но его можно включить в манифест сборки с помощью параметра `AddModules`. Если в рамках одной процедуры компиляции создается несколько модулей, типы `internal` из одного модуля будут доступны для других модулей, компилируемых вместе с ним. Если код одного модуля ссылается на типы `internal` в другом модуле, оба модуля нужно включить в манифест сборки с помощью параметра `internal`.

2.2.2.9. winexe

Параметр `winexe` указывает компилятору создать исполняемый файл (EXE) программы. Исполняемый файл создается с расширением EXE. Программа предоставляет пользовательский интерфейс либо из библиотеки .NET, либо с помощью API-интерфейсов Windows. Воспользуйтесь параметром `exe`, чтобы создать консольное приложение. Если не указано иное с помощью параметра `OutputAssembly`, имя выходного файла совпадает с именем входного файла, который содержит метод. В файлах исходного кода, который компилируется в EXE-файл, должен содержаться один и только один метод `Main`. Если код содержит несколько классов с методом, указать, какой класс содержит метод, можно с помощью параметра `StartupObject`.

2.2.2.10. winmdobj

Если используется параметр `winmdobj`, компилятор создает промежуточный WINMDOBJ-файл, который можно преобразовать в двоичный WINMD-файл среды выполнения. Затем WINMD-файл можно использовать в программах на языках JavaScript и C++ в дополнение к программам, использующим управляемые языки.

Параметр `winmdobj` сигнализирует компилятору, что необходим промежуточный модуль. затем winmdobj-файл можно подать через средство экспорта, чтобы создать файл метаданных (winmd). WINMD-файл содержит код из исходной библиотеки и метаданные WinMD, используемые JavaScript, C++ и средой выполнения. Выходные данные файла, скомпилированного с помощью параметра компилятора `winmdobj`, используются только в качестве входных данных инструментом экспорта WimMDExp (на WINMDOBJ-файл нет прямой ссылки). Выходной файл получает имя первого входного файла, если только не используется параметр `OutputAssembly`. Метод `Main` не требуется.

2.2.2.11. appcontainerexe

Если используется параметр компилятора `appcontainerexe`, компилятор создает исполняемый файл (EXE-файл), который должен запускаться в контейнере приложения. Этот параметр аналогичен `-target:winexe`, но предназначен для приложений магазина Windows 8.x.

Этот параметр устанавливает бит в переносимом исполняемом файле (PE), чтобы обеспечить запуск приложения в контейнере. Если он установлен, при попытке запустить исполняемый файл вне контейнера приложения методом `CreateProcess` будет возникать ошибка. Выходной файл получает имя входного файла, содержащего метод, если только с помощью параметра `OutputAssembly` не указано иное.

2.2.3. Параметры ввода

Следующие параметры управляют входными данными компилятора.

`References` / `-reference` or `-references`

Ссылка на метаданные из указанного файла или файлов сборки.

AddModules / **-addmodule**

Добавление модуля, созданного с помощью `target:module` для этой сборки.

EmbedInteropTypes / **-link**

Внедрение метаданных из указанных файлов сборки взаимодействия.

2.2.3.1. References

Параметр `Reference` указывает компилятору импортировать сведения типа `public` из указанного файла в текущий проект. Это позволяет ссылаться на метаданные из указанных файлов сборки.

```
<Reference Include="filename" />
```

`filename` – это имя файла, который содержит манифест сборки. Чтобы импортировать данные из нескольких файлов, включите отдельный элемент `Reference` для каждого файла. Псевдоним можно определить как дочерний элемент элемента `Reference`:

```
<Reference Include="filename.dll">
    <Aliases>LS</Aliases>
</Reference>
```

В предыдущем примере `LS` является допустимым идентификатором C#, представляющим корневое пространство имен, которое будет содержать все пространства имен в сборке `LS`. Импортируемые файлы должны содержать манифест. Для указания каталога, в котором находятся одна или несколько ссылок на сборки, используйте `AdditionalLibPaths`. В разделе с описанием `AdditionalLibPaths` также рассматриваются каталоги, в которых компилятор ищет сборки. Чтобы компилятор мог распознавать тип в сборке (не в модуле), ему следует указать принудительно разрешать типы. Это можно сделать, определив экземпляр типа. Возможны и другие способы разрешения компилятором имен типов в сборке. Например, если тип наследуется от типа в сборке, его имя будет распознаваться компилятором. Иногда бывает необходимо сослаться на две различные версии одного компонента из одной сборки. Для этого используйте элемент `Aliases` в элементе `References` для каждого файла, чтобы различать два этих файла. Этот псевдоним используется в качестве квалификатора имени компонента и разрешается в компонент в одном из файлов.

2.2.3.2. AddModules

Установка этого параметра приводит к добавлению модуля, созданного с помощью параметра `<TargetType>module</TargetType>` для текущей компиляции:

```
<AddModule Include=file1 />
<AddModule Include=file2 />
```

Где `file`, `file2` – это выходные файлы, содержащие метаданные. В этот файл не может входить манифест сборки. Чтобы импортировать несколько файлов, разделите их имена запятыми или точками с запятой. Все модули, добавленные с помощью `AddModules`, во время выполнения должны находиться в том же каталоге, что и выходной файл. То есть во время компиляции можно ука-

зать модуль в любом каталоге, но во время выполнения он должен находиться в каталоге приложения. Если во время выполнения модуль отсутствует в каталоге приложения, возникнет исключение `TypeLoadException`. `file` не может содержать сборку. Например, если выходной файл был создан с помощью параметра модуля `TargetType`, для импорта его метаданных можно использовать `AddModules`.

Если выходной файл был создан с помощью параметра `TargetType`, отличающегося от модуля, для импорта его метаданных нельзя использовать `AddModules`, но можно использовать параметр `References`.

2.2.3.3. EmbedInteropTypes

Дает компилятору указание сделать всю информацию о типах COM из указанных сборок доступной компилируемому проекту.

```
<References>
    <EmbedInteropTypes>file1;file2;file3</EmbedInteropTypes>
</References>
```

Где `file1;file2;file3` – список имен файлов сборок, разделенных точкой с запятой. Если имя файла содержит пробел, заключите его в кавычки. Параметр `EmbedInteropTypes` позволяет развернуть приложение, содержащее сведения о внедренном типе. После этого приложение может использовать типы из сборки среды выполнения, реализующей информацию о внедренных типах, без ссылки на эту сборку. Если опубликовано несколько версий сборки среды выполнения, приложение, содержащее сведения о внедренных типах, может работать с различными версиями без перекомпиляции.

Параметр `EmbedInteropTypes` особенно полезен при работе с COM-взаимодействием. COM-типы внедряются для того, чтобы приложению не требовалась основная сборка взаимодействия (PIA) на целевом компьютере. Параметр `EmbedInteropTypes` указывает компилятору внедрить сведения о COM-типах из указанной сборки взаимодействия в результирующий скомпилированный код. COM-тип определяется значением CLSID (GUID). Это позволяет запускать приложение на целевом компьютере, где установлены те же COM-типы с такими же значениями CLSID. В качестве примера можно привести приложения, автоматизирующие Microsoft Office. Поскольку в приложениях типа Office значение CLSID обычно не зависит от версии, ваше приложение сможет использовать COM-типы по ссылке до тех пора, пока на целевом компьютере установлена платформа .NET Framework 4 или более поздней версии, а приложение работает с методами, свойствами или событиями, включенными в эти COM-типы. Параметр `EmbedInteropTypes` внедряет только интерфейсы, структуры и делегаты. Внедрение COM-классов не поддерживается.

Примечание. Если в коде создается экземпляр внедренного COM-типа, его следует создавать, используя соответствующий интерфейс. При попытке создать экземпляр внедренного COM-типа с помощью компонентного класса возникнет ошибка.

Как и параметр `References` компилятора, параметр `EmbedInteropTypes` компилятора использует файл ответов Csc.rsp, который ссылается на часто используемые сборки .NET. Если вы не хотите, чтобы компилятор использовал файл Csc.rsp, примените параметр `NoConfig` компилятора.

```
// The following code causes an error if ISampleInterface is an embedded interop type.
```

```
ISampleInterface<SampleType> sample;
```

Типы с универсальным параметром, тип которого внедрен из сборки взаимодействия, нельзя использовать, если он относится к внешней сборке. Это ограничение не относится к интерфейсам. Например, рассмотрим интерфейс `Range`, который определен в сборке `Microsoft.Office.Interop.Excel`. Если библиотека содержит внедренные типы взаимодействия из сборки `Microsoft.Office.Interop.Excel` и предоставляет метод, возвращающий универсальный тип с параметром, типом которого является интерфейс `Range`, этот метод должен возвращать универсальный интерфейс, как показано в следующем примере кода.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Office.Interop.Excel;

public class Utility
{
    // The following code causes an error when called by a client assembly.
    public List<Range> GetRange1()
    {
        return null;
    }

    // The following code is valid for calls from a client assembly.
    public IList<Range> GetRange2()
    {
        return null;
    }
}
```

В следующем примере клиентский код может вызывать метод, возвращающий универсальный интерфейс IList без ошибок.

```
public class Client
{
    public void Main()
    {
        Utility util = new Utility();

        // The following code causes an error.
        List<Range> rangeList1 = util.GetRange1();

        // The following code is valid.
        List<Range> rangeList2 = (List<Range>)util.GetRange2();
    }
}
```

2.2.4. Параметры ошибок и предупреждений

Следующие параметры управляют тем, как компилятор сообщает об ошибках и предупреждениях.

`WarningLevel` / `-warn`

Определение порога предупреждений.

`AnalysisLevel`

Задаёт необязательный уровень предупреждений.

`TreatWarningsAsErrors` / **`-warnaserror`**

Обработка всех предупреждений как ошибок.

`WarningsAsErrors` / **`-warnaserror`**

Обработка одного или нескольких предупреждений как ошибок.

`WarningsNotAsErrors` / **`-warnnotaserror`**

Обработка одного или нескольких предупреждений не как ошибок.

`DisabledWarnings` / **`-nowarn`**

Определение списка отключенных предупреждений.

`CodeAnalysisRuleSet` / **`-ruleset`**

Определение файла набора правил, который отключает определенную диагностику.

`ErrorLog` / **`-errorlog`**

Определение файла для регистрации всей диагностики компилятора и анализатора.

`ReportAnalyzer` / **`-reportanalyzer`**

Отчет о дополнительных сведениях анализатора, таких как время выполнения.

2.2.4.1. WarningLevel

Параметр `WarningLevel` определяет порог предупреждений для отображения компилятором.

```
<WarningLevel>3</WarningLevel>
```

Значение элемента – это порог предупреждений, который должен отображаться для компиляции: при меньших значениях отображаются только самые серьезные предупреждения. Чем выше значение, тем больше предупреждений будет отображаться. Значение должно быть больше нуля или равно ему.

Уровень предупреждений	Значение
0	Отключает вывод всех предупреждающих сообщений
1	Отображает серьезные предупреждающие сообщения
2	Отображает предупреждения уровня 1, а также некоторые менее серьезные предупреждения, в том числе связанные со скрытием членов класса
3	Отображает предупреждения уровня 2, а также некоторые менее серьезные предупреждения, в том числе связанные с выражениями, которые всегда возвращают значение `true` или `false`
4 (по умолчанию)	Отображает все предупреждения уровня 3, а также информационные предупреждения

Предупреждение. Командная строка компилятора принимает значения, превышающие 4, чтобы включить предупреждения о волне предупреждения. Однако пакет SDK.NET задает WarningLevel в соответствии с AnalysisLevel в файле проекта.Используйте TreatWarningsAsErrors,

чтобы обрабатывать все предупреждения как ошибки. Используйте DisabledWarnings, чтобы отключить определенные предупреждения.

2.2.4.2. Analysis level

Analysis level	Значение
5	Отображает все необязательные предупреждения волны 5
6	Отображает все необязательные предупреждения волны 6
7	Отображает все необязательные предупреждения волны 7
latest (по умолчанию)	Отображает все информационные предупреждения вплоть до текущего выпуска
preview	Отображает все информационные предупреждения вплоть до последней предварительной версии
none	Отключает все информационные предупреждения

Чтобы получить сведения об ошибке или предупреждении, выполните поиск по соответствующему коду в указателе справочной системы. Используйте `TreatWarningsAsErrors`, чтобы обрабатывать все предупреждения как ошибки. Используйте `DisabledWarnings`, чтобы отключить определенные предупреждения.

2.2.4.3. TreatWarningsAsErrors

Параметр `TreatWarningsAsErrors` включает обработку всех предупреждений как ошибок. Вы также можете использовать `TreatWarningsAsErrors`, чтобы обрабатывать как ошибки только некоторые предупреждения. Если включить `TreatWarningsAsErrors`, можно использовать `WarningsNotAsErrors` для перечисления предупреждений, которые не следует рассматривать как ошибки.

```
<TreatWarningsAsErrors>true</TreatWarningsAsErrors>
```

Все предупреждающие сообщения вместо этого будут выводиться как ошибки. Процесс сборки будет остановлен (выходные файлы не будут создаваться). По умолчанию `TreatWarningsAsErrors` не применяется, и это означает, что наличие предупреждений не препятствует созданию выходного файла. Если требуется обрабатывать как ошибки только конкретные предупреждения, укажите их номера через запятую. Набор всех предупреждений о допустимости значений `NULL` можно указать с помощью сокращения `Nullable`. Используйте `WarningLevel`, чтобы определить порог предупреждений, которые будет отображать компилятор. Используйте `DisabledWarnings`, чтобы отключить определенные предупреждения.

Важно. Существует два незначительных различия между использованием <TreatWarningsAsErrors> элемента в csproj-файле и использованием warnaserror параметра командной строки MSBuild. TreatWarningsAsErrors влияет только на компилятор C#, а не на другие задачи MSBuild в csproj-файле. Параметр командной строки warnaserror влияет на все задачи. Во-вторых,

компилятор не выдает никаких выходных данных при использовании TreatWarningsAsErrors. Компилятор выдает выходные данные при использовании параметра командной строки warnaserror.

2.2.4.4. WarningsAsErrors и WarningsNotAsErrors

Параметры `WarningsAsErrors` и `WarningsNotAsErrors` переопределяют параметр `TreatWarningsAsErrors` для списка предупреждений. Этот параметр можно использовать со всеми предупреждениями CS. Префикс CS является необязательным. Можно использовать число или cs, за которым следует уровень ошибки или предупреждения.

Включение предупреждений 0219 и 0168 как ошибок:

```
<WarningsAsErrors>0219,CS0168</WarningsAsErrors>
```

Отключение этих же предупреждений как ошибок:

```
<WarningsNotAsErrors>0219,CS0168</WarningsNotAsErrors>
```

Вы можете использовать `WarningsAsErrors`, чтобы настроить набор предупреждений как ошибок. Используйте `WarningsNotAsErrors`, чтобы настроить набор предупреждений, которые не должны обрабатываться как ошибки, если вы включили обработку всех предупреждений как ошибок.

2.2.4.5. DisabledWarnings

Параметр `DisabledWarnings` позволяет отключить отображение одного или нескольких предупреждений компилятором. Разделяйте предупреждения запятыми.

```
<DisabledWarnings>number1, number2</DisabledWarnings>
```

`number1, number2` – номера предупреждений, которые требуется отключить в компиляторе. Вы можете указать только числовую часть идентификатора предупреждения. Например, чтобы отключить CS0028, можно указать `<DisabledWarnings>28</DisabledWarnings>`. Компилятор просто пропустит номера предупреждений, переданные в `DisabledWarnings`, которые были действительными в предыдущих выпусках, но были удалены.

Следующие предупреждения нельзя отключить с помощью параметра `DisabledWarnings`:

— предупреждение компилятора (уровень 1) CS2002;

— предупреждение компилятора (уровень 1) CS2023;

— предупреждение компилятора (уровень 1) CS2029.

2.2.4.6. CodeAnalysisRuleSet

Укажите файл набора правил, который настраивает определенную диагностику.

```
<CodeAnalysisRuleSet>MyConfiguration.ruleset</CodeAnalysisRuleSet>
```

`MyConfiguration.ruleset` – это путь к файлу набора правил.

2.2.4.7. ErrorLog

Укажите файл для записи данных диагностики компилятора и анализатора в журнал.

```
<ErrorLog>compiler-diagnostics.sarif</ErrorLog>
```

Параметр `ErrorLog` указывает компилятору вывести журнал в формате обмена результатами статического анализа (SARIF). Журналы SARIF обычно считываются средствами, которые анализируют результаты диагностики компилятора и анализатора.

Формат SARIF можно указать с помощью аргумента `version` для элемента `EversionErrorLog`:

```
<ErrorLog>logVersion21.json,version=2.1</ErrorLog>
```

Разделитель может быть запятой (,) или точкой с запятой (;). Допустимые значения: 1, 2 и 2.1. Значение по умолчанию – 1. 2 и 2.1 оба означают SARIF версии 2.1.0.

2.2.4.8. ReportAnalyzer

Включение в отчет дополнительных сведений об анализаторе, включая время выполнения.

```
<ReportAnalyzer>true</ReportAnalyzer>
```

Параметр ReportAnalyzer указывает компилятору создавать дополнительные сведения журнала MSBuild, в которых подробно описываются характеристики производительности анализаторов в сборке. Обычно он используется создателями анализаторов в ходе проверки анализатора.

Важно. Дополнительные сведения журнала, созданные этим флагом, создаются только при использовании параметра -verbosity:detailed. Дополнительные сведения см. в статье swtiches в документации на MSBuild.

2.2.5. Параметры создания кода

Следующие параметры управляют созданием кода в компиляторе.

`DebugType` / `-debug`

Выдача (или невыдача) отладочных сведений.

`Optimize` / `-optimize`

Включение оптимизаций.

`Deterministic` / `-deterministic`

Создание эквивалентных (побайтовое сравнение) выходных данных для одного источника входных данных.

`ProduceOnlyReferenceAssembly` / `-refonly`

Создание базовой сборки вместо полной сборки в качестве основного выходного файла.

2.2.5.1. DebugType

При использовании параметра `DebugType` компилятор создает отладочные сведения и помещает их в выходном файле или файлах. Отладочные сведения добавляются по умолчанию для конфигурации сборки *отладки*. По умолчанию этот параметр отключен для конфигурации сборки *выпуска*.

```
<DebugType>pdbonly</DebugType>
```

Для всех версий компилятора, начиная с C# 6.0, разницы между `pdbonly` и `full` нет. Выберите `pdbonly`.

Допустимы следующие значения.

Параметр	Значение
`full`	Выводить информацию об отладке в PDB-файл, используя стандартный формат для текущей платформы: Windows: PDB-файл Windows. Linux/macOS: переносимый PDB-файл
`pdbonly`	Эквивалентно `full`. Дополнительные сведения см. в примечании ниже
`portable`	Выводить информацию об отладке в PDB-файл, используя кроссплатформенный формат переносимого PDB-файла
`embedded`	Выводить информацию об отладке в файл *DLL* или *EXE* (PDB-файл не создается), используя формат переносимого PDB-файла

Важно. Следующая информация касается только компиляторов версии ниже C# 6.0. Значение этого элемента может быть либо full, либо pdbonly. Аргумент full, действующий при отсутствии pdbonly, позволяет присоединить отладчик к выполняющейся программе. Определение pdbonly позволяет выполнять отладку исходного кода при запуске программы в отладчике, но при этом ассемблер отображается только при подключении выполняющейся программы к отладчику. Используйте этот параметр для создания отладочных сборок. При использовании параметра full нужно учитывать некоторое влияние на скорость и размер оптимизированного кода JIT и незначительное влияние на качество кода с full. Для создания кода выпуска рекомендуется использовать pdbonly или не использовать PDB. Разница между pdbonly и full заключается в том, что компилятор с full создает DebuggableAttribute, чтобы сообщить JIT-компилятору о доступности отладочных сведений. Поэтому если при использовании DebuggableAttribute код содержит DebuggableAttribute со значением false, вы получите сообщение об ошибке.

2.2.5.2. Optimize

Параметр `Optimize` включает или отключает оптимизацию кода компилятором, чтобы сделать код более быстрым, коротким и эффективным. Параметр `Optimize` включен по умолчанию для конфигурации сборки выпуска. По умолчанию этот параметр отключен для конфигурации сборки отладки.

```
<Optimize>true</Optimize>
```

Параметр `Optimize` включает оптимизацию кода во время выполнения в общеязыковой среде выполнения. По умолчанию оптимизация отключена. Чтобы включить оптимизацию, укажите `Optimize+` . При создании модуля для сборки используйте те же настройки `Optimize`, что и для сборки. Параметры `Optimize` и `Debug` можно комбинировать.

2.2.5.3. Deterministic

Указывает компилятору на необходимость создания сборки, чьи побайтовые выходные данные идентичны в разных компиляциях, если входные данные идентичны.

```
<Deterministic>true</Deterministic>
```

По умолчанию выходные данные компилятора из заданного набора входных данных являются уникальными, так как компилятор добавляет метку времени и идентификатор `MVID`, который создается из случайных чисел. Вы можете использовать параметр `<Deterministic>` для создания детерминированной сборки, двоичное содержимое которой идентично в разных компиляциях при условии, что входные данные не изменяются. В такой сборке поля `timestamp` и `MVID` будут заменены значениями, полученными из хэша всех входных данных компиляции. Компилятор рассматривает следующие входные данные, которые влияют на детерминированность:

- Последовательность параметров командной строки.
- Содержимое RSP-файла ответов в компиляторе.
- Точная версия компилятора и его связанные сборки.
- Текущий путь к каталогу.
- Двоичное содержимое всех файлов, явным образом переданных компилятору прямо или косвенно, в том числе:
  - исходные файлы;
  - связанные сборки;
  - связанные модули;
  - ресурсы;
  - файл ключа строгого имени;
  - файлы ответов @;
  - анализаторы;
  - наборы правил;
  - другие файлы, которые могут использоваться анализаторами.
- Текущий язык и региональные параметры (для языка сообщений о диагностике и исключениях).
- Кодировка по умолчанию (или текущая кодовая страница), если кодировка не указана.
- Наличие, отсутствие и содержимое файлов на пути поиска компилятора (задается, например, с помощью `-lib` или `-recurse`).
- Платформа общеязыковой среды выполнения (CLR), на которой выполняется компилятор.

— Значение `%LIBPATH%`, которое может повлиять на загрузку зависимостей анализатора.

Детерминированную компиляцию можно использовать, чтобы определить, компилируются ли двоичные данные из надежного источника. Детерминированные выходные данные могут быть полезны, если источник является общедоступным. Также с их помощью можно определить, будут ли шаги сборки, зависящие от изменений в двоичном файле, использоваться в процессе сборки.

2.2.5.4. ProduceOnlyReferenceAssembly

Параметр `ProduceOnlyReferenceAssembly` указывает на то, что в качестве основных выходных данных должна быть выведена базовая сборка, а не сборка реализации. Параметр `ProduceOnlyReferenceAssembly` автоматически отключает вывод PDB, так как базовые сборки не могут выполняться.

```
<ProduceOnlyReferenceAssembly>true</ProduceOnlyReferenceAssembly>
```

Базовые сборки являются особым типом сборки, которая содержит только минимальный объем метаданных, необходимый для представления общедоступного API библиотеки. Такие сборки включают в себя объявления для всех элементов, которые важны при указании ссылки на сборку в средствах сборки, но исключают все реализации элементов, а также объявления закрытых элементов, не имеющих наблюдаемого влияния на их контракт API.

Параметры `ProduceOnlyReferenceAssembly` и `ProduceReferenceAssembly` являются взаимоисключающими.

2.2.6. Параметры безопасности

Следующие параметры управляют параметрами безопасности компилятора.

`PublicSign` / **`-publicsign`**

Публичная подпись сборки.

`Delaysign` / **`-delaysign`**

Отложенная подпись сборки с использованием только открытой части ключа строгого имени.

`KeyFile` / **`-keyfile`**

Указывает файл ключа строгого имени.

`KeyContainer` / **`-keycontainer`**

Указывает файл ключа строгого имени.

`HighEntropyVA` / **`-highentropyva`**

Включение случайного расположения макета адресного пространства с высокой энтропией (ASLR).

2.2.6.1. PublicSign

Этот параметр указывает компилятору на необходимость применения открытого ключа, но фактически не подписывает сборку. Кроме того, параметр `PublicSign` задает в сборке бит, тем самым сообщая среде выполнения, что файл подписан.

```
<PublicSign>true</PublicSign>
```

С параметром PublicSign необходимо использовать параметр KeyFile или KeyContainer. Параметры KeyFile и KeyContainer определяют открытый ключ. Параметры PublicSign и DelaySign – взаимоисключающие. При таком подписывании в сборку добавляется открытый ключ. Кроме того, в сборке устанавливается флаг «подписано». Такой подход иногда называют «фиктивным подписыванием» или «подписыванием OSS». Но фактически сборка не подписывается закрытым ключом. Разработчики используют публичное подписывание для проектов с открытым кодом. Им нужно создавать сборки, которые совместимы с выпущенными «полностью подписанными» сборками, но у них нет доступа к закрытому ключу, который использовался для подписывания сборок. Так как потребителям редко нужно проверять, полностью ли подписана сборка, создаваемые сообществами сборки можно использовать практические во всех случаях, в которых может использоваться полностью подписанная сборка.

2.2.6.2. DelaySign

Этот параметр указывает компилятору зарезервировать пространство в выходном файле, чтобы впоследствии добавить в него цифровую подпись.

```
<DelaySign>true</DelaySign>
```

Если требуется полностью подписанная сборка, используйте параметр DelaySign-. Если нужно лишь поместить в сборку открытый ключ, используйте параметр DelaySign. Параметр DelaySign не действует, если не использовать его с KeyFile или KeyContainer. Параметры KeyContainer и PublicSign являются взаимоисключающими. При запросе полностью подписанной сборки компилятор хэширует файл, содержащий манифест (метаданные сборки), и подписывает хэш закрытым ключом. Эта операция предназначена для создания цифровой подписи, которая хранится в файле, содержащем манифест. Если применяется отложенное подписывание сборки, компилятор не вычисляет и не сохраняет подпись, но он резервирует пространство в файле, чтобы подпись можно было добавить позже. Параметр DelaySign позволяет поместить сборку в глобальный кэш при тестировании. После тестирования можно полностью подписать сборку, поместив в нее закрытый ключ с помощью компоновщика сборок.

2.2.6.3. KeyFile

Задает имя файла, содержащего криптографический ключ.

```
<KeyFile>filename</KeyFile>
```

filename – это имя файла, содержащего ключ строгого имени. При использовании этого параметра компилятор вставляет открытый ключ из указанного файла в манифест сборки и затем подписывает окончательную сборку закрытым ключом. Чтобы создать файл ключа, в командной строке введите sn -k file. При компиляции с параметром -target:module имя файла ключа сохраняется в модуле и включается в сборку, создаваемую при компиляции с параметром AddModules. Также можно передать сведения о шифровании компилятору с помощью параметра KeyContainer. Если

требуется использовать частично подписанную сборку, укажите параметр DelaySign. Если для одной процедуры компиляции одновременно заданы параметры KeyFile и KeyContainer, сначала будет предпринята попытка использовать контейнер ключей. В случае успеха сборка подписывается данными контейнера ключей. Если компилятору не удалось обнаружить контейнер ключей, будет предпринята попытка использовать файл, заданный параметром KeyFile. В случае успеха сборка подписывается данными из файла ключей, и эти данные о ключах будут помещены в контейнер ключей. Таким образом при следующей компиляции контейнер ключей будет действителен. Файл ключей может содержать только открытый ключ.

2.2.6.4. KeyContainer

Задает имя контейнера криптографического ключа.

```
<KeyContainer>container</KeyContainer>
```

container – это имя контейнера ключа строгого имени. Если используется параметр KeyContainer, компилятор создает совместно используемый компонент. Компилятор вставляет открытый ключ из указанного контейнера в манифест сборки, после чего подписывает финальную сборку закрытым ключом. Чтобы создать файл ключа, в командной строке введите sn -k file. Команда sn -i устанавливает пару ключей в контейнер. Этот параметр не поддерживается, если компилятор работает в CoreCLR. Чтобы подписать сборку при компиляции в CoreCLR, используйте параметр KeyFile. При компиляции с параметром TargetType имя файла ключа сохраняется в модуле и включается в сборку при компиляции этого модуля с параметром AddModules. Этот параметр можно также указать в исходном коде любого модуля MSIL в качестве настраиваемого атрибута (System.Reflection.AssemblyKeyNameAttribute). Также можно передать сведения о шифровании компилятору с помощью параметра KeyFile. Используйте delaysign, чтобы добавить открытый ключ в манифест сборки, но подписывание сборки до ее тестирования.

2.2.6.5. HighEntropyVA

Параметр компилятора HighEntropyVA сообщает ядру операционной системы, поддерживает ли указанный исполняемый файл технологию Address Space Layout Randomization (ASLR) с высокой энтропией.

```
<HighEntropyVA>true</HighEntropyVA>
```

Этот параметр указывает, что 64-битный исполняемый файл или исполняемый файл, отмеченный параметром компилятора PlatformTarget, поддерживает виртуальный диапазон адресов с высокой энтропией. Этот параметр отключен по умолчанию. Чтобы включить его, используйте параметр HighEntropyVA.

Параметр HighEntropyVA позволяет совместимым версиям ядра операционной системы использовать более высокие степени энтропии во время рандомизации размещения диапазона адресов в рамках ASLR. Использование более высоких степеней энтропии означает, что можно выделить больше адресов таким областям памяти, как стеки и кучи. Из-за этого сложнее подобрать расположение определенной области памяти. Если указан параметр компилятора HighEntropyVA, целевой

исполняемый файл и все модули, от которых он зависит, должны иметь возможность обработать значения указателя, размер которых превышает 4 гигабайта (ГБ), если они выполняются как 64-разрядный процесс.

2.2.7. Параметры ресурсов

Следующие параметры определяют способ создания и импорта ресурсов Win32 компилятором C#.

`Win32Resource` / **`-win32res`**

Указание файла ресурсов Win32 (RES-файл).

`Win32Icon` / **`-win32icon`**

Создание ссылки на метаданные из указанного файла или файлов сборки.

`Win32Manifest` / **`-win32manifest`**

Указание файла манифеста Win32 (XML-файл).

`NoWin32Manifest` / **`-nowin32manifest`**

Указание не включать манифест Win32 по умолчанию.

`Resources` / **`-resource`**

Встраивание указанного ресурса (короткая форма: `/res`).

`LinkResources` / **`-linkresources`**

Связывание указанного ресурса с этой сборкой.

2.2.7.1. Win32Resource

Параметр `Win32Resource` вставляет ресурс Win32 в выходной файл.

```
<Win32Resource>filename</Win32Resource>
```

`filename` – это файл ресурсов, который требуется добавить в выходной файл. Ресурс Win32 может содержать сведения о версии или точечный рисунок (значок) для упрощения идентификации приложения в проводнике. Если этот параметр не задан, компилятор будет создавать сведения о версии на основе версии сборки.

2.2.7.2. Win32Icon

Параметр `Win32Icon` вставляет в выходной файл ICO-файл, который придает выходному файлу необходимый вид в проводнике.

```
<Win32Icon>filename</Win32Icon>
```

`filename` – это файл с расширением *.ico*, который требуется добавить в выходной файл. Файл с расширением *.ico* можно создать с помощью компилятора ресурсов. Компилятор ресурсов вызывается при компиляции программы Visual C++. При этом файл с расширением *.ico* создается из файла *.rc*.

2.2.7.3. Win32Manifest

Параметр `Win32Manifest` позволяет указать пользовательский файл манифеста приложения win32manifest для внедрения в переносимый исполняемый файл проекта (PE-файл).

```
<Win32Manifest>filename</Win32Manifest>
```

`filename` – это имя и расположение пользовательского файла манифеста. По умолчанию компилятор C# внедряет манифест приложения, определяющий запрошенный уровень выполнения `asInvoker`. Он создает манифест в той же папке, в которой создается исполняемый файл. Если необходимо предоставить пользовательский манифест, например, чтобы задать уровень выполнения `highestAvailable` или `requireAdministrator`, используйте этот параметр, чтобы указать имя файла.

Примечание. Этот параметр и параметр Win32Resources являются взаимоисключающими. При попытке использовать оба параметра в одной командной строке возникнет ошибка построения.

Приложение требует виртуализации в любом из следующих случаев:

— Можно использовать параметр `NoWin32Manifest` и не предоставлять манифест на более позднем этапе сборки или в файле ресурсов (*.res*) с помощью параметра `Win32Resource`.

— Можно предоставлять пользовательский манифест, который не определяет запрошенный уровень выполнения.

csc создает файл по умолчанию с расширением *.manifest* и сохраняет его в каталогах отладки и выпуска вместе с исполняемым файлом. Пользовательский манифест можно добавить, создав его в любом текстовом редакторе и добавив полученный файл в проект. Вы также можете щелкнуть значок проекта в «Обозревателе решений» и нажать кнопку «Добавить новый элемент», а затем «Файл манифеста приложения». Добавленный новый или существующий файл манифеста появится в раскрывающемся списке «Манифест».

Манифест приложения можно предоставить во время пользовательского этапа после сборки или в составе файла ресурсов Win32 с помощью параметра `NoWin32Manifest`.

2.2.7.4. NoWin32Manifest

С помощью параметра `NoWin32Manifest` можно указать компилятору не внедрять манифест приложения в исполняемый файл.

```
<NoWin32Manifest />
```

2.2.7.5. Resources

Внедряет указанный ресурс в выходной файл.

```
<Resources Include=filename>
    <LogicalName>identifier</LogicalName>
    <Access>accessibility-modifier</Access>
</Resources>
```

filename – это файл ресурсов .NET, который требуется внедрить в выходной файл. identifier (необязательно) – это логическое имя ресурса, используемое для его загрузки. По умолчанию используется имя файла. accessibility-modifier (необязательно) – это доступность ресурса: public (открытый) или private (закрытый). Значение по умолчанию: public. По умолчанию ресурсы в сборке открыты, если они создавались с помощью компилятора C#. Чтобы сделать ресурс закрытым, укажите параметр private в качестве модификатора доступа. Уровни доступности, отличные от public или private, не допускаются. Если filename является файлом ресурсов .NET, созданным, например, с помощью filename или в среде разработки, то к нему можно обращаться с помощью элементов в пространстве имен System.Resources. Для получения дополнительной информации см. System.Resources.ResourceManager. Чтобы получить доступ ко всем остальным ресурсам во время выполнения, используйте методы GetManifestResource в классе Assembly. Порядок расположения ресурсов в выходном файле будет определяться порядком, указанным в файле проекта.

2.2.7.6. LinkResources

Создает в выходном файле ссылку на ресурс .NET. Файл ресурсов не добавляется в выходной файл. Параметр LinkResources отличается от параметра Resource, который внедряет файл ресурсов в выходной файл.

```
<LinkResources Include=filename>
    <LogicalName>identifier</LogicalName>
    <Access>accessibility-modifier</Access>
</LinkResources>
```

filename – это файл ресурсов .NET, ссылку на который необходимо создать из сборки. identifier (необязательно) – это логическое имя ресурса, используемое для его загрузки. По умолчанию используется имя файла. accessibility-modifier (необязательно) – это доступность ресурса: public (открытый) или private (закрытый). Значение по умолчанию: public. По умолчанию связанные ресурсы в сборке открыты, если они создавались с помощью компилятора C#. Чтобы сделать ресурс закрытым, укажите параметр private в качестве модификатора доступа. Модификаторы, отличные от public или private, не допускаются. Если filename является файлом ресурсов .NET, созданным, например, с помощью filename или в среде разработки, то к нему можно обращаться с помощью элементов в пространстве имен System.Resources. Для получения дополнительной информации см. System.Resources.ResourceManager. Чтобы получить доступ ко всем остальным ресурсам во время выполнения, используйте методы GetManifestResource в классе Assembly. Файл, указанный в параметре filename, может иметь любой формат. Например, может потребоваться сделать имеющуюся на компьютере библиотеку DLL частью сборки, поэтому ее можно разместить в глобальном кэше сборок и обеспечить к ней доступ из управляемого кода сборки. Это действие можно также выполнить в компоновщике сборок.

2.2.8. Прочие параметры

Следующие параметры управляют прочей функциональностью компилятора.

`ResponseFiles` / **`-@`**

Считывание файла ответов с дополнительными параметрами.

`NoLogo` / **`-nologo`**

Запрещает отображение сообщения компилятора об авторских правах.

`NoConfig` / **`-noconfig`**

запрещает автоматическое включение файла *CSC.RSP*.

2.2.8.1. ResponseFiles

С помощью параметра `ResponseFiles` можно указать файл, содержащий параметры компилятора и список файлов исходного кода, которые требуется компилировать.

```
<ResponseFiles>response_file</ResponseFiles>
```

`response_file` указывает файл, содержащий список параметров компилятора и файлов исходного кода, которые требуется компилировать. Параметры компилятора и файлы исходного кода будут обрабатываться компилятором таким образом, как если бы они были указаны в командной строке. Чтобы задать несколько файлов ответов для компиляции, используйте соответствующее число параметров файла ответов. В одной строке файла ответов может содержаться несколько параметров компилятора и файлов исходного кода. Спецификация отдельного параметра компилятора должна размещаться на одной строке и не может разбиваться на несколько строк. В файл ответов можно добавлять комментарии, которые должны начинаться с символа #. Указание параметров компилятора в файле ответов аналогично выполнению соответствующих команд из командной строки. Компилятор обрабатывает параметры команд в том порядке, в котором они считываются. Аргументы командной строки могут переопределять параметры, заданные ранее в файле ответов. Аналогичным образом, параметры в файле ответов будут переопределять параметры, ранее заданные в командной строке или в других файлах ответов. В C# представлен файл `csc.rsp`, который находится в одном каталоге с файлом **csc**. Дополнительные сведения о формате файла ответов см. в разделе о параметре NoConfig. Этот параметр компилятора нельзя задать в среде разработки Visual Studio или изменить программными средствами. Ниже приведено несколько строк из образца файла ответов:

```
 # build the first output file
-target:exe -out:MyExe.exe source1.cs source2.cs
```

2.2.8.2. NoLogo

Параметр `NoLogo` отключает отображение приветствия при запуске компилятора и информационных сообщений во время компиляции.

```
<NoLogo>true</NoLogo>
```

2.2.8.3. NoConfig

Параметр `NoConfig` указывает компилятору не использовать файл *csc.rsp* при компиляции.

```
<NoConfig>true</NoConfig>
```

Файл *csc.rsp* содержит ссылки на все сборки, поставляемые вместе с .NET Framework. Фактические ссылки, которые включает среда разработки Visual Studio .NET, зависят от типа проекта. Вы можете изменить файл *csc.rsp* и указать дополнительные параметры компилятора, которые нужно включать при каждой компиляции. Если не требуется, чтобы компилятор искал и использовал параметры в файле *csc.rsp*, укажите параметр `NoConfig`. Этот параметр компилятора недоступен в Visual Studio и не может быть изменен программным способом.

2.2.9. Расширенные параметры

Следующие параметры поддерживают сложные сценарии.

`MainEntryPoint, StartupObject` / **`-main`**

Указание типа, который содержит точку входа.

`PdbFile` / **`-pdb`**

Указание имени файла с данными отладки.

`PathMap` / **`-pathmap`**

Указание сопоставления для вывода компилятором имен исходных путей.

`ApplicationConfiguration` / **`-appconfig`**

Указание файла конфигурации приложения, который содержит параметры привязки сборки.

`AdditionalLibPaths` / **`-lib`**

Указание дополнительных каталогов для поиска ссылок.

`GenerateFullPaths` / **`-fullpath`**

Компилятор будет создавать абсолютные пути.

`PreferredUILang` / **`-preferreduilang`**

Указание имени предпочтительного языка для вывода данных.

`BaseAddress` / **`-baseaddress`**

Указание базового адреса библиотеки для сборки.

`ChecksumAlgorithm` / **`-checksumalgorithm`**

Указание алгоритма для расчета контрольной суммы файла источника, хранящегося в PDB.

`CodePage` / **`-codepage`**

Указание кодовой страницы, используемой при открытии исходных файлов.

`Utf8Output` / **`-utf8output`**

Сообщения компилятора будут выводиться в кодировке UTF-8.

`FileAlignment` / **`-filealign`**

Указание выравнивания для разделов выходного файла.

`ErrorEndLocation` / **`-errorendlocation`**

Выходные строка и столбец конечного расположения каждой ошибки.

`NoStandardLib` / **`-nostdlib`**

Не указывать ссылку на стандартную библиотеку *mscorlib.dll*.

`SubsystemVersion` / **`-subsystemversion`**

Указание версии подсистемы для этой сборки.

`ModuleAssemblyName` / `-moduleassemblyname`
Имя сборки, частью которой будет этот модуль.

2.2.9.1. MainEntryPoint или StartupObject

Этот параметр определяет класс, который содержит точку входа в программу, если метод `Main` содержит сразу несколько классов.

```
<StartupObject>MyNamespace.Program</StartupObject>
```

или

```
<MainEntryPoint>MyNamespace.Program</MainEntryPoint>
```

`Program` – это тип, содержащий метод `Main`. Указанное имя класса должно быть полным; оно должно включать полное пространство имен, содержащее ключевое слово `class`, за которым следует имя класса. Например, если метод `Main` находится в классе `Program` в пространстве имен `MyApplication.Core`, необходимо указать параметр компилятора `-main:MyApplication.Core.Program`. Если ваша компиляция включает более одного типа с методом `Main`, вы можете указать, какой тип содержит метод `Main`.

Примечание. Этот параметр нельзя использовать для проекта, который включает инструкции верхнего уровня, даже если этот проект содержит один или несколько методов Main.

2.2.9.2. PdbFile

Параметр компилятора `PdbFile` задает имя и расположение файла отладочных символов. Значение `filename` указывает на имя и расположение файла отладочных символов.

```
<PdbFile>filename</PdbFile>
```

Если указан `DebugType`, компилятор создаст PDB-файл в том же каталоге, в котором он создаст выходной файл (EXE или DLL). PDB-файл имеет такое же базовое имя файла, что и выходной файл. С помощью параметра `PdbFile` можно задать имя и расположение PDB-файла, отличающееся от используемых по умолчанию. Этот параметр компилятора нельзя задать в среде разработки Visual Studio или изменить программными средствами.

2.2.9.3. PathMap

Параметр компилятора `PathMap` определяет способ сопоставления компилятором физических путей и выходных имен исходных путей. Этот параметр сопоставляет каждый физический путь на компьютере, где выполняется компилятор, с соответствующим путем, который должен быть записан в выходные файлы. В следующем примере `path1` – это полный путь к исходным файлам в текущей среде, а `sourcePath1` – исходный путь, подставляемый вместо `path1` во всех выходных файлах. Чтобы указать несколько сопоставленных исходных путей, разделите их точкой с запятой.

```
<PathMap>path1=sourcePath1; path2=sourcePath2</PathMap>
```

Компилятор записывает исходный путь в выходные данные по следующим причинам:

1. Исходный путь подставляется вместо аргумента, когда `CallerFilePathAttribute` применяется как необязательный параметр.
2. Исходный путь внедряется как PDB-файл.
3. Путь к PDB-файлу внедряется в PE-файл (переносимый исполняемый файл).

2.2.9.4. ApplicationConfiguration

Параметр компилятора `ApplicationConfiguration` позволяет приложению C# задать расположение файла конфигурации приложения сборки (`app.config`) в среде CLR во время привязки сборки.

```
<ApplicationConfiguration>file</ApplicationConfiguration>
```

`file` – это файл конфигурации приложения, содержащий параметры привязки сборки. Один из случаев использования параметра `ApplicationConfiguration` – сложные сценарии, когда в сборке одновременно используются ссылки на версию .NET Framework и на версию .NET Framework для Silverlight определенной базовой сборки. Например, для конструктора XAML, написанного в Windows Presentation Foundation (WPF), может потребоваться ссылаться на оба рабочих стола WPF, для пользовательского интерфейса конструктора и для подмножества WPF, поставляемого с Silverlight. Одна и та же сборка конструктора имеет доступ к обеим сборкам. По умолчанию отдельные ссылки вызывают ошибку компиляции, так как привязка сборки видит две эквивалентные сборки. Параметр компилятора `ApplicationConfiguration` позволяет указать расположение файла app.config, который отключает поведение по умолчанию с помощью тега `<supportPortability>`, как показано в следующем примере.

```
<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

Компилятор передает расположение файла в логику с привязкой сборки среды CLR.

Примечание. Чтобы использовать файл app.config, уже заданный в проекте, добавьте тег свойства <UseAppConfigForCompiler> в CSPROJ-файл и задайте для него значение true. Чтобы указать другой файл app.config, добавьте тег свойства <AppConfigForCompiler> и задайте в качестве его значения расположение требуемого файла.

В следующем примере показан файл app.config, позволяющий приложению иметь ссылки на реализации .NET Framework и .NET Framework для реализации Silverlight любой сборки .NET Framework, существующей в обеих реализациях. Параметр компилятора `ApplicationConfiguration` позволяет указать расположение этого файла app.config.

```
<configuration>
  <runtime>
    <assemblyBinding>
      <supportPortability PKT="7cec85d7bea7798e" enable="false"/>
      <supportPortability PKT="31bf3856ad364e35" enable="false"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

2.2.9.5. AdditionalLibPaths

Параметр AdditionalLibPaths позволяет задать расположение сборок, указанных с помощью параметра References.

```
<AdditionalLibPaths>dir1[,dir2]</AdditionalLibPaths>
```

dir1 – это каталог, в котором компилятор должен искать базовую сборку, если она отсутствует в текущем рабочем каталоге (из которого был вызван компилятор) и системном каталоге среды CLR. dir2 – это один или несколько дополнительных каталогов для поиска связанных сборок. Имена каталогов разделяются запятыми без пробелов. Компилятор выполняет поиск ссылок на сборки, для которых не указано полное имя, в следующем порядке:

1. Текущая рабочая папка.
2. Системный каталог среды CLR.
3. Каталоги, указанные с помощью параметра AdditionalLibPaths.
4. Каталоги, указанные переменной среды LIB.

Для указания ссылки на сборку используется параметр Reference. AdditionalLibPaths является аддитивным, то есть каждое следующее указание этого параметра присоединяется к предыдущим значениям. Так как путь к зависимой сборке не указывается в манифесте сборки, приложение найдет используемую сборку в глобальном кэше сборок. Компилятор, ссылающийся на сборку, не подразумевает, что среда CLR может найти и загрузить сборку во время выполнения.

2.2.9.6. GenerateFullPaths

Параметр GenerateFullPaths указывает компилятору на необходимость задавать полный путь к файлу при выводе списка ошибок и предупреждений компиляции.

```
<GenerateFullPaths>true</GenerateFullPaths>
```

По умолчанию для ошибок и предупреждений, возникающих в ходе компиляции, указывается имя файла, в котором они были обнаружены. Параметр GenerateFullPaths указывает компилятору на необходимость задать полный путь к файлу. Этот параметр компилятора недоступен в Visual Studio и не может быть изменен программным способом.

2.2.9.7. PreferredUILang

С помощью параметра компилятора PreferredUILang можно указать язык, на котором компилятор C# отображает выходные данные, например сообщения об ошибках.

```
<PreferredUILang>language</PreferredUILang>
```

language – это название языка, который будет использоваться для вывода компилятора. Параметр компилятора PreferredUILang можно использовать, чтобы указать язык, который компилятор C# должен использовать для сообщений об ошибках и других данных вывода командной строки.

Если необходимый языковой пакет не установлен, вместо него используются языковые настройки операционной системы.

2.2.9.8. BaseAddress

Параметр `BaseAddress` позволяет указать предпочтительный базовый адрес для загрузки библиотеки DLL.

```
<BaseAddress>address</BaseAddress>
```

`address` – это базовый адрес для библиотеки DLL. Этот адрес можно задать в десятичном, шестнадцатеричном или восьмеричном формате. Базовый адрес по умолчанию для библиотеки DLL задается в среде выполнения .NET. Младшее слово этого адреса будет округляться, например значение `0x11110001` округляется до `0x11110000`. Чтобы завершить процесс подписи для библиотеки DLL, используйте файл `SN` с параметром `-R`.

2.2.9.9. ChecksumAlgorithm

Этот параметр управляет алгоритмом контрольной суммы, который используется для кодирования исходных файлов в формат PDB.

```
<ChecksumAlgorithm>algorithm</ChecksumAlgorithm>
```

Параметр `algorithm` должен иметь значение `SHA1` (по умолчанию) или `SHA256`.

2.2.9.10. CodePage

Этот параметр позволяет указать кодовую страницу, используемую во время компиляции, если требуемая страница не является текущей кодовой страницей системы по умолчанию.

```
<CodePage>id</CodePage>
```

`id` – это идентификатор кодовой страницы, используемой для всех файлов исходного кода при компиляции. Во-первых, компилятор будет пытаться интерпретировать все исходные файлы в кодировке UTF-8. Если кодировка файлов исходного кода отличается от UTF-8 и использует символы, отличные от 7-разрядных символов ASCII, используйте параметр `CodePage`, чтобы указать нужную кодовую страницу. Параметр `CodePage` применяется ко всем файлам исходного кода, включенным в компиляцию.

2.2.9.11. Utf8Output

Параметр `Utf8Output` отображает выходные данные компилятора в кодировке UTF-8.

```
<Utf8Output>true</Utf8Output>
```

В некоторых конфигурациях для различных языков выходные данные компилятора некорректно отображаются в консоли. Используйте `Utf8Output` и перенаправьте выходные данные компилятора в файл.

2.2.9.12. FileAlignment

Параметр `FileAlignment` позволяет указать размер разделов в выходном файле. Допустимые значения: 512, 1024, 2048, 4096 и 8192. Эти значения указаны в байтах.

```
<FileAlignment>number</FileAlignment>
```

Каждый раздел выравнивается по границе, кратной значению `FileAlignment`. Фиксированный размер по умолчанию не предусмотрен. Если значение `FileAlignment` не указано, среда CLR выбирает значение по умолчанию во время компиляции. Указанный размер раздела влияет на размер выходного файла. Изменение размера раздела может применяться для программ, выполняющихся на небольших устройствах.

2.2.9.13. ErrorEndLocation

Указывает компилятору выводить строку и столбец конечного расположения каждой ошибки.

```
<ErrorEndLocation>true</ErrorEndLocation>
```

По умолчанию компилятор записывает начальное расположение в исходном коде для всех ошибок и предупреждений. Если этот параметр имеет значение True, компилятор записывает как начальное, так и конечное расположения для каждой ошибки и предупреждения.

2.2.9.14. NoStandardLib

Параметр `NoStandardLib` запрещает импорт библиотеки `mscorlib.dll`, которая определяет все пространство имен `System`.

```
<NoStandardLib>true</NoStandardLib>
```

Используйте этот параметр, если вы хотите определить или создать собственное пространство имен `System` и объекты. Если вы не укажете параметр `NoStandardLib`, библиотека `mscorlib.dll` будет импортирована в вашу программу (как и при указании `<NoStandardLib>false</NoStandardLib>`).

2.2.9.15. SubsystemVersion

Задает минимальную версию подсистемы, в которой может выполняться исполняемый файл. Чаще всего этот параметр предоставляет исполняемому файлу возможность использовать функции безопасности, недоступные в прежних версиях операционной системы.

Примечание. Чтобы указать саму подсистему, используйте параметр компилятора TargetType.<SubsystemVersion>major.minor</SubsystemVersion>major.minor задает минимальную требуемую версию подсистемы (через точку записываются основная и дополнительная версии). Например, можно указать, что приложение не может работать в операционной системе старше Windows 7. Установите для этого параметра значение 6.01, как описано в таблице ниже в этой статье. Укажите значения для параметров major и minor в виде целых чисел. Нули в начале версии minor не изменяют версию, нули в конце – изменяют. Например, 6.1 и 6.01 – одна версия, а 6.10 – другая. Рекомендуется указывать дополнительный номер версии двумя цифрами, чтобы избежать путаницы.

Значение по умолчанию параметра компилятора SubsystemVersion зависит от условий в следующем списке:

— `/target:appcontainerexe`

— `/target:winmdobj`

— `-platform:arm`

— Значение по умолчанию – 6.02, если задан любой параметр компилятора из следующего списка.

— Значение по умолчанию – 6.00, если используется средство MSBuild, приложение предназначено для .NET Framework 4.5 и не установлены параметры компилятора, определенные ранее в этом списке.

— Если ни одно из предыдущих условий не верно, значение по умолчанию – 4.00.

2.2.9.16. ModuleAssemblyName

Указывает имя сборки, к отдельным типам которой может обращаться *.netmodule*.

```
<ModuleAssemblyName>assembly_name</ModuleAssemblyName>
```

Параметр `ModuleAssemblyName` нужно использовать при сборке *.netmodule* и выполнении следующих условий:

— Для *.netmodule* требуется доступ к неоткрытым типам в существующей сборке.

— Известно имя сборки, в которой будет создан *.netmodule*.

— Существующая сборка предоставила дружественной сборке доступ к сборке, в которую будет встроен *.netmodule*.

Дополнительные сведения о сборке *.netmodule* см. в разделе о параметре `TargetType` и его значении `module`.

2.3. Ассемблер промежуточного кода

Ассемблер IL создает переносимый исполняемый (PE) файл из сборки на промежуточном языке (IL). Можно запустить полученный исполняемый файл, содержащий код на промежуточном языке и необходимые метаданные, чтобы проверить, выполняется ли код IL так, как ожидалось.

В командной строке введите следующее:

`ilasm [options] имя файла [[options]имя файла...]`

Аргумент `имя файла` – это имя исходного файла с расширением IL. В этом файле содержатся директивы объявления метаданных и символические инструкции IL. Программа *Ilasm* может создать один PE-файл из нескольких исходных файлов, для чего следует указать несколько аргументов исходных файлов.

Примечание. Убедитесь, что последняя строка кода в исходном IL-файле имеет либо конечный пробел, либо символ конца строки.

`/32bitpreferred`

Создает предпочтительно 32-разрядный образ (PE32).

`/alignment:integer`

Параметр `integer` задает значение «FileAlignment» в необязательном заголовке NT. Если в файле указана IL-директива ALIGNMENT, этот параметр ее переопределяет.

`/appcontainer`

Создает файл *DLL* или *EXE*, выполняющийся в контейнере приложения в качестве выходных данных.

`/arm`

Задает ARM как целевой процессор. Если разрядность образа не задана, в качестве значения по умолчанию используется `/32bitpreferred`.

`/Baseinteger`

Параметр `integer` задает значение «ImageBase» в необязательном заголовке NT. Если в файле указана IL-директива IMAGEBASE, этот параметр ее переопределяет.

`/clock`

Измеряет и выводит следующие значения времени компиляции указанного исходного IL-файла в миллисекундах. `Total Run` – общее время, затраченное на перечисленные ниже операции. `Startup` – загрузка и открытие файла. `Emitting MD` – порождение метаданных. `Ref to Def Resolution` – разрешение ссылок на определения в файле. `CEE File Generation` – создание образа файла в памяти. `PE File Writing` – запись образа в PE-файл

`/Debug[:Impl|OPT]`

Включает отладочные сведения (имена локальных переменных и аргументов, номера строк). Создает PDB-файл. `/debug` без дополнительных значений отключает JIT-оптимизацию и использует точки последовательности из PDB-файла. без дополнительных значений отключает JIT-оптимизацию и использует точки последовательности из PDB-файла. `IMPL` отключает JIT-оптимизацию и использует неявные точки последовательности. `OPT` включает JIT-оптимизацию и использует неявные точки последовательности.

`/dll`

Выходным файлом будет библиотека *DLL*.

`ENCfile`

Создает разности «Изменить и продолжить» из указанного файла источника. Данный аргумент предназначен только для использования в учебных заведениях и не поддерживается для коммерческого использования.

`/exe`

Выходным файлом будет исполняемый файл. Это значение по умолчанию.

/Flagsinteger

Параметр `integer` задает значение «ImageFlags» в заголовке среды CLR. Если в файле указана IL-директива CORFLAGS, этот параметр ее переопределяет. Список допустимых значений параметра *integer*см. в CorHdr.h, COMIMAGE_FLAGS.

/fold

Свертывает идентичные тела методов в один.

/highenvarlistentryopyva

На выходе создает исполняемый файл, который поддерживает технологию Address Space Layout Randomization (ASLR) с высокой энтропией. Используется по умолчанию для `/appcontainer`.

/includeincludePath

Задает путь для поиска файлов, включенных с помощью команды

`#include`

/KeykeyFile

Компилирует файл `filename` со строгой подписью с помощью закрытого ключа в `keyFile`.

/key: @keySource

Компилирует файл `filename` со строгой подписью с помощью закрытого ключа, созданного в `keySource`.

/listing

Создает файл списка со стандартными выходными данными. Если этот параметр не задан, файл списка не создается. Этот параметр не поддерживается в .NET Framework 2.0 и более поздних версиях.

/mdv:versionString

Задает строку версии метаданных.

/msv:major .minor

Задает версию потока метаданных, где `major` и `minor` являются целыми числами.

/noautoinherit

Отключает наследование по умолчанию из класса Object, если базовый класс не указан.

/nocorstub

Запрещает создание заглушки CORExeMain.

/nologo

Отключает отображение эмблемы Майкрософт при запуске.

/Outputfile.ext

Задает имя и расширение выходного файла. По умолчанию имя выходного файла совпадает с именем первого исходного файла. Расширение по умолчанию – *EXE*. Если задан параметр `/dll`, по умолчанию используется расширение *DLL*.

Примечание. Задание параметра /output:myfile.dll не равносильно указанию параметра /dll. Если параметр /dll не задан, будет создан исполняемый файл с именем myfile.dll.

/optimize

Оптимизирует длинные инструкции в короткие. Например, `br` в `br.s`.

/pe64

Создает 64-разрядный образ (PE32+).

/pdb

Создает PDB-файл, отслеживание отладочной информации не включается.

/quiet

Задает тихий режим и отключает вывод сведений о ходе сборки.

/Resourcefile.res

Включает указанный файл ресурсов в формате *. res в результирующем *.exe* или *.dll* файле. С параметром `/resource` может быть указан только один RES-файл.

/ssver:int .int

Задает номер версии подсистемы в необязательном заголовке NT. Для `/appcontainer` и `/arm` минимальным номером версии является 6.02.

/StackstackSize

Задает `stackSize`в качестве значения «SizeOfStackReserve» в необязательном заголовке NT.

/stripreloc

Указывает, что перемещения базового адреса не требуются.

/Subsysteminteger

Параметр `integer` задает значение «subsystem» в необязательном заголовке NT. Если в файле указана IL-директива SUBSYSTEM, этот параметр ее переопределяет. Список допустимых значений параметра `integer` см. в winnt.h, IMAGE_SUBSYSTEM.

/x64

Задает 64-разрядный процессор AMD в качестве целевого процессора. Если разрядность образа не задана, в качестве значения по умолчанию используется `/pe64`.

/?

Отображает синтаксис команд и параметров программы.

Примечание. Параметры программы Ilasm не учитывают регистр и распознаются по первым трем буквам. Например, /lis равнозначно /listing, а /res: myresfile.res равнозначно /resource: myresfile.res. Разделителем параметра и его аргумента может служить двоеточие (:) или знак равенства (=). Например, /output:file.ext эквивалентно /output=file.ext.

2.3.1. Примечания

Используя программу *Ilasm.exe*, компилятор и другие средства, разработчики могут сосредоточить свои усилия на работе с IL и создании метаданных, а не на преобразовании IL в формат PE-файла.

Аналогично таким компиляторам для среды выполнения, как C# и Visual Basic, программа *Ilasm* не создает промежуточные объектные файлы и при создании PE-файла позволяет пропустить этап связывания.

Ассемблер IL может выразить все существующие метаданные и возможности IL языков программирования, предназначенные для взаимодействия со средой выполнения. С его помощью можно адекватно выразить на ассемблере IL и скомпилировать с помощью программы *Ilasm.exe* управляемый код, написанный на любом из этих языков.

Примечание. Компиляция может завершиться ошибкой, если последняя строка кода в исходном IL-файле не имеет конечного пробела или символа конца строки.

Программа *Ilasm* может применяться совместно с сопутствующей программой – Ildasm. Программа *Ildasm* анализирует PE-файл, содержащий IL-код, и создает текстовый файл, подходящий

для обработки программой *Ildasm*. Это полезно, к примеру, при компиляции кода на языке программирования, не поддерживающем все атрибуты метаданных среды выполнения. После компиляции кода и обработки результатов с помощью программы *Ildasm* можно вручную добавить недостающие атрибуты в полученный текстовый IL-файл. Чтобы создать итоговый исполняемый файл, этот текстовый файл следует обработать с помощью программы *Ilasm.exe*.

Эту технологию можно также использовать для создания одного PE-файла из нескольких PE-файлов, созданных различными компиляторами.

Чтобы обеспечить наибольшую точность совместной работы программ *Ildasm.exe* и *Ilasm.exe*, по умолчанию ассемблер не заменяет короткие коды на длинные, которые могут быть в источниках IL (или созданы другим компилятором). С помощью параметра **/optimize** можно заменить короткие коды там, где это возможно.

2.3.2. Сведения о версии

Начиная с .NET Framework 4.5 к реализации интерфейса можно добавить свой атрибут с помощью кода, похожего на следующий:

```
.class interface public abstract auto ansi IMyInterface
{
  .method public hidebysig newslot abstract virtual
    instance int32 method1() cil managed
  {
  } // end of method IMyInterface::method1
} // end of class IMyInterface
.class public auto ansi beforefieldinit MyClass
  extends [mscorlib]System.Object
  implements IMyInterface
  {
    .interfaceimpl type IMyInterface
    .custom instance void
      [mscorlib]System.Diagnostics.DebuggerNonUserCodeAttribute::.ctor() = ( 01 00 00 00 )
      …
```

Начиная с .NET Framework 4.5 можно указать произвольный большой двоичный объект (BLOB) маршала с помощью необработанного двоичного представления, как показано в следующем коде:

```
.method public hidebysig abstract virtual
        instance void
        marshal({ 38 01 02 FF })
        Test(object A_1) cil managed
```

Дополнительные сведения о грамматике IL см. в файле asmparse.grammar в составе Windows SDK.

2.3.2.1. Примеры

Следующая команда выполняет сборку IL-файла *myTestFile.il* и создает исполняемый файл *myTestFile.exe*.

```
ilasm myTestFile
```

Следующая команда выполняет сборку IL-файла *myTestFile.il* и создает *DLL*-файл *myTestFile.dll*.

```
ilasm myTestFile /dll
```

Следующая команда выполняет сборку IL-файла *myTestFile.il* и создает *DLL*-файл *myNewTestFile.dll*.

```
ilasm myTestFile /dll /output:myNewTestFile.dll
```

В приведенном ниже примере кода показано очень простое приложение, выводящее сообщение «Hello World!» на консоль. Можно скомпилировать этот код и создать IL-файл с помощью программы *Ildasm.exe* [https://docs.microsoft.com/ru-ru/dotnet/framework/tools/ildasm-exe-il-disassembler].

```
using System;

public class Hello
{
    public static void Main ( String[] args )
    {
        Console.WriteLine( "Hello World!");
    }
}
```

Следующий пример кода IL соответствует предыдущему примеру кода на C#. Можно скомпилировать этот код в сборку с помощью ассемблера IL. В обоих примерах кода (на IL и C#) на консоли отображается «Hello World!».

```
// Metadata version: v2.0.50215
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\V.4..
  .ver 2:0:0:0
}
.assembly sample
{
  .custom instance void [mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAt
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module sample.exe
// MVID: {A224F460-A049-4A03-9E71-80A36DBBBCD3}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x02F20000

// =============== CLASS MEMBERS DECLARATION ==================

.class public auto ansi beforefieldinit Hello
       extends [mscorlib]System.Object
{
  .method public hidebysig static void  Main(string[] args) cil managed
  {
    .entrypoint
```

```
    // Code size       13 (0xd)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr      "Hello World!"
    IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
  } // end of method Hello::Main

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method Hello::.ctor

} // end of class Hello
```

2.4. Компилятор ресурсов

Генератор файлов ресурсов (Resgen.exe) преобразует текстовые файлы (TXT или RESTEXT) и файлы ресурсов на основе XML (RESX) в двоичные файлы среды CLR (RESOURCES), которые можно внедрить в двоичный исполняемый файл среды выполнения или вспомогательную сборку.

Программа resgen.exe представляет собой универсальную программу для преобразования ресурсов, которая выполняет следующие задачи:

- Преобразование TXT-файлов и RESTEXT-файлов в файлы RESOURCES или RESX. (Формат RESTEXT-файлов идентичен формату файлов .txt. Однако расширение restext помогает легко определять текстовые файлы, содержащие определения ресурсов.
- Преобразование RESOURCES-файлов в RESX-файлы или текстовые файлы.
- Преобразование RESX-файлов в RESOURCES-файлы или текстовые файлы.
- Извлекает строковые ресурсы из сборки в RESW-файл, который пригоден для использования в приложении Магазина Windows 8.x.
- Создает класс со строгой типизацией, который предоставляет доступ к ресурсам с отдельными именами и экземпляру ResourceManager.

В случае любого сбоя программа resgen.exe возвращается значение «–1».

Чтобы получить справочные сведения о синтаксисе команд и параметрах программы Resgen.exe, можно ввести следующую команду без параметров:

```
resgen
```

Также можно использовать ключ `/?`.

```
resgen /?
```

При использовании программы Resgen.exe для создания двоичных RESOURCES-файлов можно воспользоваться языковым компилятором, чтобы внедрить двоичные файлы в исполняемые сборки, или Компоновщиком сборок (Al.exe), чтобы скомпилировать их во вспомогательные сборки.

2.4.1. Синтаксис

```
resgen  [-define:symbol1[,symbol2,...]]
[/useSourcePath] filename.extension | /compile filename.extension...
[outputFilename.extension]
[/r:assembly] [/str:lang[,namespace[,class[,file]]] [/publicclass]]

resgen filename.extension [outputDirectory]
```

2.4.2. Параметры

`/define:`*symbol1 [, Symbol2,...]*

Начиная с .NET Framework 4.5 поддерживается условная компиляция в файлы ресурсов на основе текста (TXT или RESTEXT). Если *symbol* соответствует символу, включенному во входной текстовый файл в конструкции `#ifdef`, соответствующий строковый ресурс включается в RESOURCES-файл. Если входной текстовый файл содержит оператор `#if !` с символом, который не определен ключом `/define`, соответствующий строковый ресурс включается в файл ресурсов.

Параметр `/define` игнорируется, если он применяется для нетекстовых файлов. В символах учитывается регистр.

Дополнительные сведения об этом параметре см. в разделе «Условная компиляция ресурсов» ниже.

`useSourcePath`

Задает использование текущего каталога входного файла для разрешения относительных путей к файлам.

`/compile`

Позволяет указать несколько RESX-файлов или текстовых файлов, которые следует преобразовать в несколько RESOURCES-файлов в ходе одной групповой операции. Если этот параметр не указывается, можно задать только один аргумент входного файла. Имена выходных файлов имеют формат *имя_файла*.resources.

Данный параметр невозможно использовать одновременно с параметром `/str:`.

Дополнительные сведения об этом параметре см. в разделе «Компиляция или преобразование нескольких файлов» ниже.

`/r: assembly`

Ссылается на метаданные из указанной сборки. Он используется при преобразовании RESX-файлов и позволяет программе Resgen.exe выполнять сериализацию или десериализацию ресурсов объектов. Он похож на параметр `/reference:` или `/r:` для компиляторов C# и Visual Basic.

`filename.extension`

Задает имя входного файла, который требуется преобразовать. Если вы используете первый вариант синтаксиса командной строки, представленный перед этой таблицей, `extension` должен быть одним из следующих:

TXT или RESTEXT

Текстовый файл, который преобразуется в файл с расширением RESOURCES или RESX. Текстовые файлы могут содержать только строковые ресурсы.

RESX

Файл ресурсов на основе XML, который преобразуется в RESOURCES-файл или текстовый файл (с расширением TXT или RESTEXT).

RESOURCES

Двоичный файл ресурсов, который преобразуется в RESX-файл или текстовый файл (с расширением TXT или RESTEXT).

Если используется второй, более короткий синтаксис командной строки, представленный перед этой таблицей, параметр `extension` должен быть одним из следующих:

EXE или DLL

Сборка .NET Framework (исполняемый файл или библиотека), строковые ресурсы которой должны извлекаться в RESW-файл для последующего использования при разработке приложений.

`outputFilename.extension`

Задает имя и тип создаваемого файла ресурсов.

При преобразовании из файлов с расширением TXT, RESTEXT или RESX в RESOURCES-файл этот аргумент необязателен. Если аргумент `outputFilename` не указан, программа Resgen.exe добавляет расширение RESOURCES к входному файлу `filename` и записывает этот файл в каталог, который содержит `filename,extension`.

Аргумент `outputFilename.extension` является обязательным при преобразовании из RESOURCES-файла. Для преобразования RESOURCES-файла в файл ресурсов на основе XML следует указать имя файла. Для преобразования RESOURCES-файла в текстовый файл следует указать имя файла с расширением TXT или RESTEXT. RESOURCES-файл можно преобразовать в TXT-файл только в том случае, если RESOURCES-файл содержит только строковые значения.

`outputDirectory`

Задает каталог, в котором будет сохранен RESW-файл, содержащий строковые ресурсы в `filename.extension`. Каталог `outputDirectory` должен существовать.

`/str: language[,namespace[,classname[,filename]]]`

Создает файл класса ресурса со строгой типизацией на языке программирования, указанном в параметре `language`. `language` может состоять из одного из следующих литералов:

- для C#: c#, cs или csharp;
- для Visual Basic: vb или visualbasic;
- для VBScript: vbs или vbscript;
- для C++: c++, mc или cpp;
- для JavaScript: js, jscript или javascript.

Параметр `namespace` задает для проекта пространство имен по умолчанию, параметр `classname` задает имя создаваемого класса, а параметр `filename` – имя файла класса.

С помощью параметра `/str:` можно указать только один входной файл, поэтому этот параметр нельзя использовать с параметром `/compile`.

Если указан параметр `namespace`, а параметр `classname` – нет, имя класса задается на основе имени выходного файла (например, точки заменяются символами подчеркивания). В результате ресурсы со строгой типизацией могут работать неправильно. Чтобы этого избежать, укажите и имя класса, и имя выходного файла.

Дополнительные сведения об этом параметре см. в разделе «Создание класса ресурсов со строгой типизацией» ниже.

`/publicClass`

Создает класс ресурса со строгой типизацией как открытый класс. По умолчанию классом ресурса в C# является `internal`, а в Visual Basic – `Friend`.

Этот параметр игнорируется, если не указан параметр `/str:`.

2.4.3. Программа resgen.exe и типы файлов ресурсов

Чтобы программа resgen.exe могла успешно выполнять преобразование, файлы ресурсов, текстовые файлы и RESX-файлы должны иметь правильный формат.

2.4.3.1. Текстовые файлы (TXT и RESTEXT)

Текстовые файлы (TXT или RESTEXT) могут содержать только строковые ресурсы. Строковые ресурсы используются при создании приложений, содержащих строки, которые будут переводиться на разные языки. Например, строки меню можно легко локализовать с помощью соответствующего строкового ресурса. Программа Resgen.exe считывает текстовые файлы, содержащие пары имя/значение, где имя представляет собой строку, описывающую ресурс, а значение является собственно строкой ресурса.

Текстовый файл, содержащий ресурсы, должен быть сохранен в кодировке UTF-8 или UTF-16, кроме тех случаев, когда он содержит только символы основной латиницы (до U+007F). Программа Resgen.exe удаляет знаки расширенного набора ANSI при обработке текстового файла, который сохранен в кодировке ANSI.

Программа Resgen.exe проверяет текстовый файл на наличие повторяющихся имен ресурсов. Если в текстовом файле содержатся совпадающие имена ресурсов, программа Resgen.exe выведет предупреждение и проигнорирует второе значение.

2.4.3.2. RESX-файлы

Файл ресурсов RESX состоит из записей XML. В этих записях XML можно указывать строковые ресурсы как в текстовых файлах. Основное преимущество RESX-файлов по сравнению с текстовыми файлами состоит в том, что в них можно также указывать или внедрять объекты. При просмотре RESX-файла внедренный объект (например, рисунок) отображается в двоичном виде, если эти двоичные данные являются частью манифеста ресурса. Как и текстовые файлы, RESX-файл можно открыть в текстовом редакторе, чтобы записывать, анализировать или иным образом обрабатывать его содержимое. Следует отметить, что для этого необходимо хорошо знать теги XML и структуру RESX-файла.

Чтобы создать RESOURCES-файл, содержащий внедренные нестроковые объекты, можно преобразовать с помощью программы Resgen.exe RESX-файл, содержащий объекты, или добавить ресурсы объектов из кода непосредственно в файл, используя методы класса `ResourceWriter`.

При использовании программы Resgen.exe для преобразования RESX-файла или RESOURCES-файла, содержащего объекты, в текстовый файл все строковые ресурсы будут пре-

образованы правильно, но при этом типы данных нестроковых объектов также будут записаны в файл в виде строк. При таком преобразовании внедренные объекты будут потеряны, а программа Resgen.exe выдаст сообщение об ошибке извлечения ресурсов.

2.4.3.3. Преобразование типов файлов ресурсов

При преобразовании разных типов файлов ресурсов программа Resgen.exe может быть не в состоянии выполнять преобразование или может терять сведения о конкретных ресурсах в зависимости от типа исходного и конечного файлов. В следующей таблице указаны типы успешных преобразований из файла ресурсов одного типа в другой.

Тип, из которого выполняется преобразование	В текстовый файл	В RESX	В RESW	В RESOURCES
Текстовый файл (TXT или RESTEXT)	—	Без проблем	Не поддерживается	Без проблем
RESX-файл	Преобразование завершается неудачей, если файл содержит нестроковые ресурсы (включая файловые связи)	—	Не поддерживается	Без проблем
RESOURCES-файл	Преобразование завершается неудачей, если файл содержит нестроковые ресурсы (включая файловые связи)	Без проблем	Не поддерживается	—
Сборка EXE или DLL	Не поддерживается	Не поддерживается	Только строковые ресурсы (включая пути) считаются ресурсами	Не поддерживается

2.4.3.4. Выполнение конкретных задач resgen.exe

Программу Resgen.exe можно использовать разными способами: компилировать файл ресурсов на основе текста или XML в двоичный файл, выполнять преобразование файлов в разные форматы, создавать класс, который образует оболочку для функциональных возможностей `ResourceManager` и предоставляет доступ к ресурсам. В данном разделе представлены подробные сведения по каждой задаче.

- Компиляция ресурсов в двоичный файл.
- Преобразование типов файлов ресурсов.
- Компиляция или преобразование нескольких файлов.
- Экспорт ресурсов в RESW-файл.
- Условная компиляция ресурсов
- Создание класса ресурсов со строгой типизацией

2.4.3.5. Компиляция ресурсов в двоичный файл

Чаще всего программа resgen.exe используется для компиляции файла ресурсов на основе текста (TXT или RESTEXT) или XML (RESX) в двоичный RESOURCES-файл. После компиляции выходной файл можно встроить в основную сборку с помощью языкового компилятора или во вспомогательную сборку с помощью компоновщика сборок (al.exe).

Синтаксис компиляции файла ресурсов выглядит следующим образом.

```
resgen inputFilename [outputFilename]
```

Используются следующие параметры.

`inputFilename` Имя компилируемого файла ресурсов, включая расширение. Программа Resgen.exe компилирует только файлы с расширениями TXT, RESTEXT или RESX.

`outputFilename` Имя выходного файла. Если не указывать `outputFilename`, программа Resgen.exe создаст RESOURCES-файл с корневым именем файла `inputFilename` в том же каталоге, что и `inputFilename`. Если `outputFilename` содержит путь к каталогу, этот каталог должен существовать.

Полное имя пространства имен для RESOURCES-файла указывается в имени файла и отделяется от корневого имени файла точкой. Например, если параметр `outputFilename` имеет значение `MyCompany.Libraries.Strings.resources`, пространство имен имеет вид `MyCompany.Libraries`.

Следующая команда считывает пары имя/значение из файла «Resources.txt» и создает двоичный RESOURCES-файл с именем «Resources.resources». Поскольку имя выходного файла не указано явным образом, его имя по умолчанию аналогично имени входного файла.

```
resgen Resources.txt
```

Следующая команда считывает пары имя/значение из файла «Resources.restext» и создает двоичный файл ресурсов с именем «StringResources.resources».

```
resgen Resources.restext StringResources.resources
```

Следующая команда считывает входной файл на основе XML «Resources.resx» и создает двоичный RESOURCES-файл с именем «Resources.resources».

```
resgen Resources.resx Resources.resources
```

2.4.3.6. Преобразование типов файлов ресурсов

Помимо компиляции файлов ресурсов на основе текста или XML в двоичные файлы ресурсов программа resgen.exe может преобразовать любой поддерживаемый тип файлов в другой. Это означает, что программа может выполнить следующие преобразования.

— TXT-файлы и RESTEXT-файлы в RESX-файлы.

— RESX-файлы в TXT-файлы и RESTEXT-файлы.

— RESOURCES-файлы в TXT-файлы и RESTEXT-файлы.

— RESOURCES-файлы в RESX-файлы.

Синтаксис аналогичен синтаксису, который показан в предыдущем разделе.

Следующая команда считывает двоичный файл ресурсов «Resources.resources» и создает выходной файл на основе XML с именем «Resources.resx».

```
resgen Resources.resources Resources.resx
```

Следующая команда считывает файл ресурсов на основе текста «StringResources.txt» и создает файл ресурсов на основе XML с именем «LibraryResources.resx». Помимо хранения строковых ресурсов RESX-файл можно также использовать для хранения нестроковых ресурсов.

```
resgen StringResources.txt LibraryResources.resx
```

Следующие две команды считывают файл ресурсов на основе XML «Resources.resx» и создают текстовые файлы с именами «Resources.txt» и «Resources.restext». Следует заметить, что если RESX-файл содержит внедренные объекты, они будут неправильно преобразованы в текстовые файлы.

```
resgen Resources.resx Resources.txt
resgen Resources.resx Resources.restext
```

2.4.3.7. Компиляция или преобразование нескольких файлов

С помощью ключа `/compile` можно преобразовать список файлов ресурсов из одного формата в другой за одну операцию. Синтаксис выглядит следующим образом.

```
resgen /compile filename.extension [filename.extension...]
```

Следующая команда компилирует три файла – «StringResources.txt», «TableResources.resw» и «ImageResources.resw» – в отдельные RESOURCES-файлы с именами «StringResources.resources», «TableResources.resources» и «ImageResources.resources».

```
resgen /compile StringResources.txt TableResources.resx ImageResources.resx
```

2.4.3.8. Экспорт ресурсов в RESW-файл

Синтаксис для создания RESW-файлов из сборки выглядит следующим образом.

```
resgen filename.extension  [outputDirectory]
```

Используются следующие параметры.

`filename.extension` – имя сборки .NET Framework (исполняемый файл или библиотека DLL). Если файл не содержит ресурсы, программа Resgen.exe не создает файлы.

`outputDirectory` – существующий каталог, в котором сохраняются RESW-файлы. Если параметр `outputDirectory` не задан, RESW-файлы записываются в текущий каталог. Программа Resgen.exe создает один RESW-файл для каждого RESOURCES-файла в сборке. Корневое имя RESW-файла аналогично корневому имени RESOURCES-файла.

Следующая команда создает RESW-файл в каталоге «Win8Resources» для каждого RESOURCES-файла, внедренного в приложение MyApp.exe.

```
resgen MyApp.exe Win8Resources
```

2.4.3.9. Условная компиляция ресурсов

Программа Resgen.exe поддерживает условную компиляцию строковых ресурсов в текстовых файлах (TXT и RESTEXT). Благодаря такому подходу можно использовать один файл ресурсов на основе текста для нескольких конфигураций построения.

В TXT- или RESTEXT-файле используется конструкция `#ifdef...#endif` для включения ресурса в файл `binary.Resources`, если определен символ, а конструкция `#if !... #endif` использует `#if !` для включения ресурса, если символ не определен. Во время компиляции символы задаются с помощью параметра `/define:`, за которым следует список разделенных запятыми символов. В сравнении учитывается регистр символов, который задается параметром `/define` и должен соответствовать регистру символов в компилируемых текстовых файлах.

Например, следующий файл с именем «UIResources.rext» содержит строковый ресурс с именем `AppTitle`, который может принимать одно из трех значений в зависимости от заданного символа: `PRODUCTION`, `CONSULT` или `RETAIL`.

```
#ifdef PRODUCTION
AppTitle=My Software Company Project Manager
#endif
#ifdef CONSULT
AppTitle=My Consulting Company Project Manager
#endif
#ifdef RETAIL
AppTitle=My Retail Store Project Manager
#endif
FileMenuName=File
```

После этого файл можно скомпилировать в двоичный RESOURCES-файл с помощью следующей команды.

```
resgen /define:CONSULT UIResources.restext
```

При этом создается RESOURCES-файл, содержащий два строковых ресурса. Значение ресурса `AppTitle` – «My Consulting Company Project Manager».

2.4.3.10. Создание класса ресурсов со строгой типизацией

Программа resgen.exe поддерживает ресурсы со строгой типизацией, при этом доступ к ресурсам инкапсулируется путем создания классов, содержащих набор статических свойств только для чтения. Это альтернативный способ прямого вызова методов класса `ResourceManager` для извлечения ресурсов. Можно включить поддержку ресурсов со строгой типизацией с помощью параметра `/str` в программе Resgen.exe, при которой реализуются функциональные возможности класса `StronglyTypedResourceBuilder`. Если задан параметр `/str`, результатом работы программы resgen.exe будет класс, содержащий свойства со строгой типизацией, которые соответствуют ресурсам, указанным входным параметром. Этот класс обеспечивает доступ только для чтения со строгой типизацией к ресурсам в отрабатываемом файле.

Синтаксис для создания ресурса со строгой типизацией выглядит следующим образом.

```
resgen inputFilename [outputFilename] /str:language[,namespace,[classname[,filename]]]
    [/publicClass]
```

Параметры и ключи.

`inputFilename` – имя файла ресурсов, включая расширение, для которого создается класс ресурса со строгой типизацией. Файл может быть двоичным RESOURCES-файлом или RESOURCES-файлом на основе текста или XML с расширением RESW, TXT, RESTEXT или RESOURCES.

`outputFilename` – имя выходного файла. Если `outputFilename` содержит путь к каталогу, этот каталог должен существовать. Если не указывать `outputFilename`, программа Resgen.exe создаст RESOURCES-файл с корневым именем файла `inputFilename` в том же каталоге, что и `inputFilename`.

Файл `outputFilename` может быть двоичным RESOURCES-файлом или файлом на основе текста или XML. Если расширение файла `outputFilename` отличается от расширения файла `inputFilename`, программа Resgen.exe выполняет преобразование файла.

Если файл `inputFilename` является RESOURCES-файлом, программа Resgen.exe копирует RESOURCES-файл, если файл `outputFilename` также является RESOURCES-файлом. Если `outputFilename` не задано, программа Resgen.exe перезаписывает файл `inputFilename` аналогичным RESOURCES-файлом.

language – язык, применяемый для создания исходного кода для класса ресурсов со строгой типизацией. Возможные значения: `cs`, `C#`, и `csharp` для кода на C#; `vb` и `visualbasic` для кода на Visual Basic; `vbs` и `vbscript` для кода на VBScript; и `c++`, `mc` и `cpp` для кода на C++.

namespace – пространство имен, содержащее класс ресурсов со строгой типизацией. RESOURCES-файл и класс ресурсов должны иметь одно и то же пространство имен. Дополнительные сведения о задании пространства имен в параметре `outputFilename` см. в разделе «Компиляция ресурсов в двоичный файл». Если параметр *namespace* не задан, класс ресурсов отсутствует в пространстве имен.

classname – имя класса ресурсов со строгой типизацией. Это имя должно совпадать с корневым именем RESOURCES-файла. Например, если программа Resgen.exe создает в RESOURCES-файл с именем «MyCompany.Libraries.Strings.resources», то именем класса ресурсов со строгой типизацией будет «String». Если параметр *classname* не задан, созданный класс является производным от корневого имени файла `outputFilename`. Если параметр `outputFilename` не задан, созданный класс является производным от корневого имени файла `inputFilename`.

Имя *classname* не должно содержать недопустимые символы, например пробелы. Если *classname* содержит внедренные пробелы или если *classname* создается по умолчанию из `inputFilename`, а `inputFilename` содержит пробелы, Resgen.exe заменяет все недопустимые символы символом подчеркивания (_).

filename – имя файла класса.

`/publicclass` делает класс ресурса со строгой типизацией открытым, а не `internal` (в C#) или `Friend` (в Visual Basic). С помощью такого решения можно оценивать ресурсы за пределами сборки, в которую они внедрены.

Класс ресурсов со строгой типизацией включает в себя следующие элементы.

— Конструктор без параметров, который можно использовать для создания экземпляра строго типизированного класса ресурсов.

— Свойство `static` (C#) или `Shared` (Visual Basic) и свойство `ResourceManager` только для чтения, которое возвращает экземпляр `ResourceManager`, который управляет ресурсом со строгой типизацией.
— Статическое свойство `Culture`, с помощью которого можно задать язык и региональные параметры, используемые для извлечения ресурсов. По умолчанию его значение равно `null`, то есть для пользовательского интерфейса используются текущий язык и региональные параметры.
— Одно свойство `static` (C#) или `Shared` (Visual Basic) и свойство только для чтения для каждого ресурса в RESOURCES-файле. Имя свойства является именем ресурса.

Например, при выполнении следующей команды выполняется компиляция файла ресурсов с именем «StringResources.txt» в «StringResources.resources» и создается класс с именем `StringResources` в файле исходного кода Visual Basic с именем «StringResources.vb», который можно использовать для доступа к диспетчеру ресурсов.

```
resgen StringResources.txt /str:vb,,StringResources
```

2.5. Дизассемблер промежуточного кода

Дизассемблер IL является сопутствующим инструментом ассемблера IL (*Ilasm*). *Ildasm* принимает переносимый исполняемый файл (PE-файл), содержащий код на промежуточном языке (IL), и создает на его основе текстовый файл, который может служить входным файлом для *Ilasm.exe*.

В командной строке введите следующее:

```
ildasm [options] [PEfilename] [options]
```

Перечисленные ниже параметры допустимы для файлов *EXE*, *DLL*, *OBJ*, *LIB* и *WINMD*.

`/out=filename`

Создает выходной файл с заданным параметром `filename` вместо вывода результатов в графический пользовательский интерфейс.

`/rtf`

Выводит данные в формате RTF. Не может использоваться с параметром `/text`.

`/text`

Отображает результаты в окне консоли вместо вывода в графический пользовательский интерфейс или выходной файл.

`/html`

Выводит данные в формате HTML. Может использоваться только с параметром `/output`.

`/?`

Отображает синтаксис команд и параметров для средства.

Перечисленные ниже дополнительные параметры допустимы для файлов *EXE*, *DLL* и *WINMD*.

`/bytes`

Отображает фактические байты в шестнадцатеричном формате в виде комментариев к инструкциям.

`/caverbal`

Создает большие двоичные объекты настраиваемых атрибутов в текстовом виде. По умолчанию задана двоичная форма.

`/linenum`

Включает ссылки на строки исходного файла.

`/nobar`

Подавляет вывод всплывающего окна с индикатором хода выполнения дизассемблирования.

`/noca`

Подавляет вывод настраиваемых атрибутов.

`/project`

Отображает метаданные в представлении для управляемого кода, а не так, как их представляет среда выполнения в машинном коде. если `PEfilename` не является файлом метаданных (`PEfilename`), этот параметр не действует.

`/pubonly`

Дизассемблирует только открытые типы и члены. Эквивалентен `/visibility:PUB`.

`/quoteallnames`

Заключает все имена в одинарные кавычки.

`/raweh`

Отображает предложения обработки исключений в исходном виде.

`/source`

Отображает строки исходного кода в виде комментариев.

`/tokens`

Отображает токены метаданных классов и членов.

`/visibility:` [`+vis…`]

Дизассемблирует только типы и члены с заданной областью видимости. Допустимы следующие значения аргумента `vis`:

`PUB` – открытый;

`PRI` – закрытый;

`FAM` – семейство;

`ASM` – сборка;

`FAA` – семейство и сборка;

`FOA` – семейство или сборка;

`PSC` – закрытая область.

Перечисленные ниже параметры допустимы для файлов *EXE*, *DLL* и *WINMD* только при выводе в файл или окно консоли.

`/all`

Задает сочетание параметров `/header`, `/bytes`, `/stats`, `/classlist` и `/tokens`.

`/classlist`

Включает список классов, определенных в этом модуле.

`/forward`

Использует прямое объявление класса.

`/headers`

Включает сведения заголовка файла в выходные данные.

`/item:` class[`::member`] [(`sig`)]

В зависимости от заданных аргументов выполняет дизассемблирование:

— Отменяет сборку указанного `class`.

— Отменяет сборку указанного `member` объекта `class` .

— Отменяет сборку `member` объекта `class` с указанной сигнатурой `sig`.

Формат `sig` выглядит следующим образом: `[instance] returnType(parameterType1, parameterType2, …, parameterTypeN)`

Примечание. В платформе .NET Framework версиях 1,0 и 1,1 после них должна следовать закрывающая круглая скобка: (sig) . В .NET Framework 2.0 и последующих версиях закрывающая скобка должна быть опущена: (sig.

`/noil`

Подавляет вывод кода сборки IL.

`/stats`

Включает статистику по образу.

`/typelist`

Создает полный список типов, чтобы сохранить упорядочение типов в круговом пути.

`/unicode`

Использует для выходных данных кодировку Юникод.

`/utf8`

Использует для выходных данных кодировку UTF-8. ANSI является значением по умолчанию.

Перечисленные ниже параметры допустимы для файлов *EXE*, *DLL*, *OBJ*, *LIB* и *WINMD* только при выводе в файл или окно консоли.

`/metadata[= specifier]`

Отображает метаданные, при этом параметр `specifier` может принимать следующие значения:

`MDHEADER` – показывать сведения и размеры заголовка метаданных;

`HEX` – показывать сведения в шестнадцатеричном и текстовом формате;

`CSV` – показывать количество записей и размеры кучи;

`UNREX` – показывать неразрешенные внешние элементы;

`SCHEMA` – показывать сведения о заголовке и схеме метаданных;

`RAW` – показывать необработанные таблицы метаданных;

`HEAPS` – показывать необработанные кучи;

`VALIDATE` – проверять согласованность метаданных.

Можно указать `/Метадата` несколько раз с разными значениями для.

Перечисленные ниже параметры допустимы для *LIB*-файлов только при выводе в файл или окно консоли.

`/objectfile=filename`

Вывод метаданных отдельного объектного файла из заданной библиотеки.

Примечание. Параметры программы Ildasm.exe не учитывают регистр и распознаются по первым трем буквам. Например, команда /quo эквивалентна команде /quoteallnames. Разделителем параметра и его аргумента может служить двоеточие (:) или знак равенства (=). Например, команда /output:filename эквивалентна команде /output=filename.

2.5.1. Примечания

Программа *Ildasm* работает только с PE-файлами, расположенными на жестком диске. Программа не обрабатывает файлы, установленные в глобальном кэше сборок.

Текстовый файл, созданный программой *Ildasm*, можно передавать в качестве входных данных в ассемблер IL (*Ilasm*). Это полезно, к примеру, при компиляции кода на языке программирования, не поддерживающем все атрибуты метаданных среды выполнения. После компиляции кода и обработки результатов с помощью *Ildasm* можно вручную добавить недостающие атрибуты в полученный текстовый файл IL. Чтобы создать окончательный исполняемый файл, следует обработать этот текстовый файл ассемблером IL.

Если программе *Ildasm* задан аргумент *имя_PE-файла*, содержащий внедренные ресурсы, будет создано несколько выходных файлов: текстовый файл с IL-кодом и RESOURCES-файл для каждого внедренного управляемого ресурса (название файла соответствует названию ресурса в метаданных). Если в аргумент *имя_PE-файла* внедрены неуправляемые ресурсы, будет создан RES-файл с именем, указанным для IL-вывода в параметре **/output**.

Примечание. Для входных файлов OBJ и LIB программа Ildasm отображает только описания метаданных. IL-код для файлов этих типов не дизассемблируется.

Чтобы определить, является ли файл EXE или *DLL* управляемым, обработайте его программой *Ildasm*. Если файл не является управляемым, программа выдаст сообщение, что у файла отсутствует допустимый заголовок среды CLR и он не может быть дизассемблирован. Если файл является управляемым, программа будет выполнена без ошибок.

2.5.2. Сведения о версии

Начиная с .NET Framework 4.5 программа *Ildasm.exe* обрабатывает нераспознанный маршалинговый объект BLOB (большой двоичный объект), отображая необработанное двоичное содержимое. В следующем примере показано, как отображается маршалинговый объект BLOB, созданный программой C#:

```
public void Test( [MarshalAs((short)70)]  int test) { }

// IL from Ildasm.exe output
.method public hidebysig instance void Test(int32  marshal({ 46 }) test) cil managed
```

Начиная с .NET Framework 4.5 программа *Ildasm* отображает атрибуты, применяемые к реализации интерфейса, как показано в следующем фрагменте выходных данных *Ildasm*:

```
.class public auto ansi beforefieldinit MyClass
  extends [mscorlib]System.Object
  implements IMyInterface
  {
    .interfaceimpl type IMyInterface
    .custom instance void
      [mscorlib]System.Diagnostics.DebuggerNonUserCodeAttribute::.ctor() = ( 01 00 00 00 )
      …
```

2.5.3. Примеры

Следующая команда вызывает отображение метаданных и декодированного кода для PE `MyHello.exe` в `MyHello.exe` графического пользовательского интерфейса по умолчанию.

```
ildasm myHello.exe
```

Следующая команда выполняет десборку файла `MyFile.exe` и сохраняет полученный текст ассемблера Il в файле `MyFile.exe`.

```
ildasm MyFile.exe /output:MyFile.il
```

Следующая команда дизассемблирует файл `MyFile.exe` и выводит выходной текст ассемблера IL в окно консоли.

```
ildasm MyFile.exe /text
```

Если файл `MyApp.exe` содержит внедренные управляемые и неуправляемые ресурсы, следующая команда создает четыре файла: `MyApp.exe`, *MyApp. Res*, *пиктограммы. Resources* и *Message. Resources*:

```
ildasm MyApp.exe /output:MyApp.il
```

Следующая команда дизассемблирует метод `MyMethod` класса `MyClass` в файле `MyFile.exe` и выводит результат в окно консоли.

```
ildasm /item:MyClass::MyMethod MyFile.exe /text
```

В предыдущем примере допустимо наличие нескольких методов с именем `MyMethod` и различными сигнатурами. Следующая команда выполняет разбор метода `MyMethod` экземпляра с типом возвращаемого значения `MyMethod` и типами параметров **Int32** и **String**.

```
ildasm /item:"MyClass::MyMethod(instance void(int32,string)" MyFile.exe /text
```

Примечание. В .NET Framework версии 1.0 и 1.1 открывающей скобке, которая следует за именем метода, должна соответствовать закрывающая скобка после сигнатуры: MyMethod(instance void(int32)). В .NET Framework 2.0 и более поздних версий закрывающая скобка должна быть опущена: MyMethod(instance void(int32).

Чтобы извлечь метод `static` (метод `Shared` в Visual Basic), следует опустить ключевое слово `instance`. Типы классов, которые не являются простыми типами (такими как `int32` и `string`), должны включать пространство имен и перед ними необходимо указывать ключевое слово `class`. Перед внешними типами должно быть указано имя соответствующей библиотеки в квадратных скобках. Следующая команда дизассемблирует статический метод с именем `MyMethod`, имеющий один параметр типа AppDomain, и возвращает значение типа AppDomain.

```
ildasm /item:"MyClass::MyMethod(class [mscorlib]System.AppDomain(
            class [mscorlib]System.AppDomain)" MyFile.exe /text
```

Перед вложенным типом необходимо указывать содержащий его класс, отделенный косой чертой (/). Например, если класс `MyNamespace.MyClass` содержит вложенный класс с именем `NestedClass`, вложенный класс указывается следующим образом: `class MyNamespace.MyClass/NestedClass`.

3. СООБЩЕНИЯ ОПЕРАТОРУ

ПК «Моно» позволяет выводить дополнительные сообщения, полезные для поиска причин проблем, возникающих при работе прикладной программы, использующую среду Моно. Для вывода дополнительных сообщений предназначен параметр командной строки `--trace=<выражение1,выражение2>`. Где выражение может быть:

all

Трассировка всех сборок.

none

Без трассировки сборок.

<программа>

Трассировка программы.

<сборка>

Трассировка сборки.

M:<тип>:<метод>

Трассировка метода.

N:<пространство имён>

Трассировка пространства имён.

T:<тип>

Трассировка типа.

-<выражение>, +<выражение>

Исключить выражение из трассировки.

Включить выражение в трассировку.

Примеры:

```
mono --trace=N:System,-T:System.Int32 program.exe
mono --trace=N:nothing program.exe
```

Вы можете включить вывод сообщений среды выполнения на стандартный вывод с помощью переменной окружения `MONO_LOG_LEVEL`. Например, `MONO_LOG_LEVEL=debug`.

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

БДУ — база данных уязвимостей

НД — нормативный документ

ОС — операционная система

ПД — программная документация

ПДС — подсистема

ПК — программный комплекс

СЗИ — средства защиты информации

СН — специальное назначение

СУБД — система управления базами данных

ЭВМ — электронная вычислительная машина

Лист регистрации изменений									
Номера листов (страниц)					Всего листов (страниц) в докум.	№ документа	Входящий № сопро-водительно-го докум. и дата	Подп.	Дата
Изм.	изме-ненных	заме-ненных	новых	аннули-рованных					