GNAT User's Guide

Supplement for the .NET Platform

AdaCore

About This Guide

This guide describes the features and the use of GNAT, the Ada development environment for the .NET platform. This guide also explains how to use the .NET API from Ada and how to interface Ada and the .NET framework.

Before reading this manual you should be familiar with the *GNAT User's Guide* as a thorough understanding of the concepts and notions explained there is needed to use GNAT effectively.

What This Guide Contains

This guide contains the following chapters:

- Chapter 1 [Getting Started with GNAT for .NET], page 3, gives an overview of GNAT and its tools and explains how to compile and run your first Ada program for the .NET platform.
- Chapter 2 [Ada & .NET Interoperability], page 5, explains how the .NET API and the services of any .NET class can be used from Ada. This section also explains how Ada services can be exported to .NET programmers.
- Chapter 3 [Using the .NET API with cil2ada], page 7, describes the `cil2ada` interfacing tool that takes any '`.dll`' file as input and generates Ada package specs as output. The resulting Ada specs can be used by Ada programs to interface to .NET.
- Chapter 4 [.NET-Specific Pragmas], page 9, explains some special pragmas that have been introduced to support certain aspects of interfacing between Ada and .NET.
- Chapter 5 [Debugging Ada Programs], page 17, describes how to run and debug Ada programs.
- Chapter 6 [Limitations], page 19, describes the language constructs, libraries and switches that are not supported by GNAT under .NET.

What You Should Know Before Reading This Guide

Before reading this document readers should be familiar with the *GNAT User's Guide* and have a conceptual understanding of the .NET technology.

Related Information

For further information about GNAT, Ada, and the .NET technology, we recommend consulting the following documents:

- *GNAT User's Guide* contains introductory and reference material for the GNAT development environment.
- *Ada 2005 Language Reference Manual* contains all reference material for the Ada 2005 programming language.

Conventions

Following are examples of the typographical and graphic conventions used in this guide:

- `source code`, and `utility program names`.

- 'Option flags'.
- 'File Names'.
- *Variables.*
- *Emphasis.*
- [optional information or parameters]
- Examples are described by text

    `and then shown this way.`

Commands that are entered by the user are preceded in this manual by the "$ " characters (dollar sign followed by space). If your system uses this sequence as a prompt, then the commands will appear exactly as you see them in the manual. If your system uses some other prompt, then the command will appear with the $ replaced by whatever prompt character you are using.

1 Getting Started with GNAT for .NET

1.1 Overview

The .NET technology, introduced by Microsoft, is a paradigm whose goal is to add platform-independent programming flexibility to applications and embedded devices such as consumer electronics, smart cards, etc.

The .NET technology consists of a comprehensive set of libraries (.NET API), and a virtual execution environment offering the same object code interface on all platforms (bytecode).

The GNAT system offers an Ada programming environment for the .NET platform. In addition to a bytecode compiler, binder and linker, GNAT contains a .NET-to-Ada-2005 binding generator that produces the Ada 2005 specs of the services contained in any .NET '`.class`' file or API.

Furthermore, because the '`.dll`' files generated by the GNAT compiler are fully compliant with the CIL standard, the user can employ any .NET debugger to debug Ada code, and can use any of the .NET tools that operate on '`.dll`' files (e.g. `ildasm`, `gacutil`, etc.).

As a side note, the GNAT system is implemented in Ada 2005 and its sources are available under the GPL.

1.2 GNAT Tools

Most tools are regular GNAT tools that have been slightly adapted for use with .NET. They are used in the same fashion as their corresponding GNAT equivalent. These tools are:

- `dotnet-gnatmake`: the GNAT automatic make program, determines the set of sources needed by an Ada compilation unit and performs the necessary build commands (to compile, bind, and link).
- `dotnet-gnat`: the GNAT project driver, calls other GNAT tools with projects.
- `dotnet-gnatcompile`: the GNAT compiler, compiles an Ada unit into one '`.il`' file. For compatibility with other platforms and some of the GNAT tools, the command `dotnet-gcc` is equivalent to `dotnet-gnatcompile`.
- `dotnet-gnatbind`: the GNAT binder, generates an Ada source file containing the elaboration code for the Ada application to run.
- `dotnet-gnatlink`: the GNAT linker, compiles the source file generated by `dotnet-gnatbind` and generates an executable (by default), or a DLL when using the '`/DLL`' switch.
- `dotnet-gnatls`: the GNAT library browser, displays information about compiled units, including dependencies on the corresponding sources files, and consistency of compilations.
- `dotnet-gnatfind`: the GNAT find utility, provides an easy way to locate the declaration and references for an Ada entity.
- `dotnet-gnatxref`: the GNAT cross-referencer, generates a full report of all cross-references in a given set of Ada units.

- `dotnet-gnatclean`: cleans up compilation artifacts.
- `dotnet-gnatelim`: eliminates uncalled subprograms.
- `dotnet-gnatmetric`: computes metrics on Ada sources.
- `dotnet-gnatname`: generates project files for your source tree.
- `dotnet-gnatpp`: produces a pretty-printed version of Ada sources.
- `dotnet-gnatprep`: performs preprocessing.
- `dotnet-gnatstub`: generates body stubs from Ada specs.
- `dotnet-gnatchop`: splits a multi-unit source file into individual files, one compilation unit per file.
- `dotnet-gnatkr`: "krunch"es GNAT names.
- `dotnet-gnatcheck`: checks coding style.

The following GNAT tools have been specifically developed for .NET:

- `cil2ada`: The GNAT interfacing tool, (see Chapter 3 [Using the .NET API with cil2ada], page 7) takes '`.dll`' files as input and generates Ada package specifications as output. The resulting Ada package specs can be `with`ed by Ada programs to interface to .NET services.

1.3 .NET Development Kits compatible with GNAT

GNAT has been tested with the .NET 2.0 framework. It may also be compatible with other frameworks, e.g. `mono` under `GNU/Linux`.

In order to use the GNAT toolset for .NET, you first need to install the .NET run-time and SDK (e.g. '`dotnetfx.exe`' and '`setup.exe`' under Windows). The run-time should come with any recent Windows version, while the SDK can generally be found either as a standalone installer or as part of Microsoft Visual Studio or Microsoft Visual Studio Express.

1.4 Compiling Your First Application with GNAT for .NET

To compile the following "Hello .NET" program put the following in file '`hello.adb`':

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Hello is
begin
   Put_Line ("Hello .NET!");
end Hello;
```

then type:

```
$ dotnet-gnatmake hello
```

This command will generate file '`hello.exe`'. To run it, just type

```
$ hello
```

To compile more complex Ada applications use `dotnet-gnatmake` as usual. If you want to use the GNAT compiler, binder, and linker separately, you will need to individually invoke the appropriate `dotnet-gnatcompile`, `dotnet-gnatbind` and `dotnet-gnatlink` commands.

2 Ada and .NET Interoperability

An attractive aspect of .NET is its growing set of API classes. It is therefore fundamental that the API be made available to the Ada programmer transparently. It is also important for the Ada programmer to be able to write libraries or APIs for the .NET platform in Ada, and that these libraries be easily usable in any .NET application. GNAT provides full interoperability between Ada and .NET.

To achieve this goal, constructs that can appear in a .NET class at the specification level are mapped to Ada either by means of a corresponding Ada feature or by using an implementation-defined Ada pragma.

In addition, the mapping from .NET to Ada is completely automated. GNAT does not include any Ada bindings for the .NET API, but instead provides a tool (see Chapter 3 [Using the .NET API with cil2ada], page 7) that can produce Ada specifications from any .NET DLL.

2.1 Importing .NET Services to Ada

To access the services provided by the .NET API or by any set of '`.dll`' files, you should proceed as follows:

1. If your .NET code is in source form (e.g. `C#`), compile it using any .NET compiler.

2. If you just want to import a variable or a subprogram from a .NET class, use `pragma Import` (see Section 4.2 [Pragma Import CIL], page 12) in the Ada code where you want to import the .NET service.

3. More generally, you can use the `cil2ada` utility to produce the Ada specs (containing the appropriate .NET-specific pragmas) for the '`.dll`', files containing the .NET services you would like to use from Ada (see Chapter 3 [Using the .NET API with cil2ada], page 7). Note that `cil2ada` preserves, in the generated Ada specs, the names of the original .NET services.

4. `with` the needed Ada specs and use their services as usual.

You can see an example of an Ada program using the .NET framework to display a window with menus and callbacks at '`<GNAT Pro installation dir>/share/examples/dotnet/CompactFramework`'.

2.2 Exporting Ada Services to .NET

To export a set of Ada services to .NET you should:

1. Use `pragma Export` and other CIL interfacing pragmas inside the Ada code (see Chapter 4 [.NET-Specific Pragmas], page 9). This gives you complete control of what is being generated and allows you to decide very precisely what the exported services look like on the .NET side.

2. Create a library containing you compiled Ada code.

   Note that there is for now no direct support of .NET libraries generation in the project files. Instead, you need to specify:

   - `-z` as builder switch (no main procedure, to perform bind/link steps even if no main is specified)

- `-n` as binder switch (no main entry point)
- `/dll` as linker switch (instruct the linker to produce a dll)
- force the generated library name to use a '.`dll`' suffix (by default, it will have a .exe extension)

For example, here is a simple project file to generate a library from all the sources used by `library.adb`:

```
project Library1 is
    for Main use ("library.adb");
    for Object_Dir use "obj";

    package Builder is
       for Executable_Suffix use ".dll";
       for Default_Switches ("ada") use ("-z");
    end Builder;

    package Binder is
       for Default_Switches ("ada") use ("-n");
    end Binder;

    package Linker is
       for Default_Switches ("ada") use ("/DLL");
    end Linker;

end Library1;
```

You can automatically create such a project from Visual Studio, by creating a new project using the 'library Ada Project' template, or you can look at the '`<GNAT Pro installation dir>/share/examples/dotnet/MixedLanguages/`' example where the '`AdaLib`' subdirectory contains a project creating an Ada library.

3. Add a reference to this library from the .NET project, and call the library elaboration routine before calling any of its services. The elaboration routine is contained in a special namespace `ada_<library_name_in_lowercase>_pkg` and is called `adainit`, e.g: `ada_lib1_pkg::adainit`. After using the library, you may also need to manually call `adafinal` to finalize any objects created on the Ada side.

4. Call the ada methods contained in the library using their underlying CIL naming scheme: all Ada names are translated to lower case, and the last package name receives a `_pkg` suffix (for example, the package Foo.Adapackage will be named in CIL `foo.adapackage_pkg`). You can easily verify the naming scheme by looking at the compiled files (with '`.il`' extension) located in the object directory: these are text files that can be read by any text editor or IDE.

You can see an example of C# program using Ada services in '`<GNAT Pro installation dir>/share/examples/dotnet/MixedLanguages/`'

3 Using the .NET API with `cil2ada`

The `cil2ada` tool takes '`.dll`' files as input and generates Ada specs as output.

3.1 Running `cil2ada`

The form of the `cil2ada` command is

    cil2ada [options] file

Where *file* is either a DLL containing .NET APIs, or an assembly name (e.g. `System.Windows.Form`, `mscorlib`, etc.). File names may be prefixed with directory information.

The output of `cil2ada` is an Ada source file for each '`class`' processed. The Ada source file contains a package spec giving the Ada declaration for the services exported by the corresponding '`class`'.

The Ada files generated are placed in the directory where the `cil2ada` command is invoked, or in the subdiretory specified via the *-o* option.

3.2 Switches for `cil2ada`

The following switches are available with the `cil2ada` utility:

`-compact`   Search assemblies from the .NET compact framework repository

`-h`   Displays the help message and exits

`-o name`   Create the files in the specified output directory.

`-q, -quiet`
              Quiet mode

`-r`   Perform also the analysis of the referenced assemblies.

`-V, --version`
              Displays the tool's version and exits

3.3 Running cil2ada on the .NET API

To be able to access the .NET API you need to use `cil2ada` to generate an Ada package spec for each public class in the API.

    $ cd some-dir
    $ cil2ada mscorlib -o bindings

This will create, in directory *some-dir/bindings*, an Ada package spec for each public .NET class included in '`mscorlib`' (the default .NET library).

4 .NET-Specific Pragmas

The simplest way to import services from .NET classes is to use the `cil2ada` tool to automatically generate the specification of the corresponding '`.class`' file. The resulting specification contains the appropriate .NET-specific pragmas.

Sometimes, however, interfacing between .NET and Ada requires more fine-grained control. For example;

- Importing just one routine into your Ada code,
- Grouping certain services from multiple '`.class`' files into a single Ada spec (for instance to provide a simplified view of the .NET API),
- Exporting Ada services to .NET.

This chapter explains the features and pragmas that are needed for full support of interfacing between .NET and Ada.

4.1 The `CIL_Constructor` Pragma

4.1.1 Background on .NET Constructors

A .NET constructor is a special method that must be invoked immediately after allocating an object, in order to initialize the object. Given the following .NET class:

```
public class C {
   public int field;
   public C ()      { field = 3; }
   public C (int i) { field = i; }
}
```

then the statement `C obj = new C (3)` accomplishes two things:

1. It allocates a new instance of class `C` in the .NET heap and sets `obj` to point to this object;

2. It then calls the constructor that takes an `int` parameter, passing `obj` to it as a hidden parameter and the value 3 for its `int` parameter.

If no constructor is provided, as in the following class:

```
class D extends C {
   float f;
}
```

then a default constructor

```
public D () {
   super ();
}
```

is automatically generated for class `D`. The call of `super()` inside this default constructor (known as a *no-arg* constructor) invokes the no-arg constructor of the superclass of `D`, that is, the constructor of class `C`.

Generally speaking, the first statement of every constructor must either be a call to another constructor of the class, or a call to a constructor of the superclass. For instance, given a constructor

```
public C (int i, int j) { this (i + j); }
```

The call `this (i + j)` invokes the constructor in class `C` that takes an `int` as its parameter. As another example, consider:

```
public D (int k) { super (k); }
```

Here `super (k)` invokes the constructor from `D`'s superclass that takes an `int` as its parameter.

Note that in both of the original constructors of class `C`, there are no calls to either `this (...)` or `super (...)`. When no such call is explicitly given, the .NET compiler automatically inserts a call to the no-arg constructor of the superclass. If (as will be explained below) the superclass does not have an accessible no-arg constructor then you must explicitly insert a call to a constructor from either the same class or its superclass.

As just noted, a class might not have an accessible no-arg constructor. This can occur only when explicit constructors are defined in the class. In this case, the no-arg constructor is not automatically generated for the class, and if a no-arg constructor is desired, you must add it explicitly. For instance, in the following class:

```
public class A {
   int ival;
   public A (int i) { ival = i; }
}

public class B extends A {
   float fval;
   public B (float f) { fval = f; }
}
```

the .NET compiler will issue a compile-time error reporting that no constructor matching `A ()` was found in class `A`, because the compiler tries to insert such a call at the beginning of `B`. To correct this problem the .NET programmer must either add a no-arg constructor `A ()` in class `A`, or else change the definition of `B`'s constructor to contain an explicit constructor, e.g., as follows:

```
public B (float f) {
  super (0);
  fval = f;
}
```

A similar situation may arise when the superclass contains a no-arg constructor that is not accessible in the subclass. For example:

```
public class A {
   int ival;
   public A (int i) { ival = i; }
   private A () { ival = 0; }
}

public class B extends A {
   float fval;
   public B (float f) { fval = f; }
}
```

This will generate the same error as above: the private no-arg constructor from `A` cannot be legally invoked from `B`, and thus the compiler's attempt to implicitly place the call `super()` as the first statement in `B`'s constructor will fail.

4.1.2 Using .NET Constructors in Ada

To map an Ada function *function-name* to a .NET constructor for some Ada *tagged-type*, GNAT provides the `CIL_Constructor` pragma. Its syntax is as follows:

```
pragma CIL_Constructor (function-name);
```

where *function-name* is the name of a function declared immediately within the same declarative part where the pragma occurs. The function must satisfy the following requirements:

- The function's result type is an access type designating a class-wide type with convention CIL declared at the same declarative level as the function (`access tagged-type`'Class`);

- The first function parameter is named `This`, and its type is a named access type designating *tagged-type*`'Class` which may have a `null` default value;

- If the constructor invokes other constructor then the first declaration in the function body is an object declaration with a default initial expression of the form `constructor-func (..., This)`, where the *constructor-func* is a `CIL_Constructor` function belonging either to *tagged-type* or to the parent type of *tagged-type*;

The effect of a `CIL_Constructor` pragma is to compile *function-name* into a constructor for the class corresponding to *tagged-type*. In addition, whenever *function-name* is invoked with a `null` value for parameter `This`, the compiler calls the `tagged-type` object allocator and passes in the pointer to the newly allocated object instead of the value `null`.

A `CIL_Constructor` pragma is a program unit pragma. It can appear in the same places where an `Inline` pragma for *function-name* can appear. The `CIL_Constructor` pragma applies to all the overloaded *function-name* subprograms declared immediately within the declarative region containing the pragma.

For examples of use of this pragma, see the packages generated by `cil2ada`.

4.1.3 .NET Constructors and Ada Allocators

If an Ada function has been defined as a no-arg constructor (via pragma `CIL_Constructor`) then it is implicitly invoked during the evaluation of an Ada allocator. For instance a client of package `C` given in the previous section could write:

```
with C;
procedure Client is
    Obj_1 : C.Ref := new_C;
    Obj_2 : C.Ref := new C.Typ;  -- allocator
```

In compiling `new C.Typ`, GNAT generates a call of the no-arg constructor if present (in the example `new_C (This : Ref := null)`). If there is no no-arg constructor then GNAT reports an error. (This last check is not supported as of September 2007, and an exception is raised at run time).

4.2 Pragma Import CIL

For convention CIL, pragma `Import` has the following syntax:

```
pragma Import ([Convention    =>] CIL,
               [Entity        =>] Local_Name
             [,[External_Name =>] String_Expression]);
```

where *Local_Name* is the name of an object, subprogram, record component, exception, or package, while *String_Expression* is a string giving the .NET name of the imported entity. If *String_Expression* is missing it is taken to be the *Local_Name*, folded to lower case.

4.2.1 Importing Packages

If the *Local_Name* of an `Import` pragma is the name of a package spec P, then all the entities declared in P must be explicitly imported from .NET. The *String_Expression* of such an `Import` pragma gives the name of the .NET class corresponding to P and can be a simple class name or it can have the form *namespace.class_name*.

The following rules apply when importing a package P:

- All the entities declared inside P must be imported either by means of the `Import` pragma or by using other .NET-specific pragmas.

- P must declare at most one tagged or untagged record type, and this type's name must be `Typ`. `Typ` models the record part of the class corresponding to P.

- P must not contain task types or protected types.

- The *String_Expression* of the `Import` pragma for an object, subprogram, or record component declared in P must be a simple name (it cannot contain any "." characters).

- Each package (if any) nested within P must itself contain an `Import` pragma (and the above rules apply recursively).

The following example illustrates these rules:

```
package MSSyst.Object is
   pragma Preelaborate;

   type Typ (<>) is tagged limited private;

   type Ref       is access all Typ;
   type Ref_Class is access all Typ'Class;

   function new_Object (This : Ref := null) return Ref;
   function Equals
     (This : access Typ;
      obj  : access MSSyst.Object.Typ'Class) return Standard.Boolean;

private
   type Typ is tagged limited null record;
   pragma Convention (CIL, Typ);

   pragma Cil_Constructor (new_Object);
   pragma Import (CIL, Equals, "Equals");

end MSSyst.Object;
pragma Import (CIL, Object, "[mscorlib]System.Object");
```

4.2.2 Importing Exceptions

If the *Local_Name* of an `Import` pragma is the name of an exception *E*, the *String_Expression* of such an `Import` pragma gives the name of the class corresponding to *E*. This can be a simple class name or it can have the form *namespace_name*.*class_name* (which says that the class *class_name* corresponding to *E* belongs to the namespace *namespace_name*).

When importing an exception you should make sure that the imported class is indeed a .NET exception, i.e. it derives from `System.SystemException`.

4.2.3 Importing Record Components

If the *Local_Name* of an `Import` pragma is the name of a record field, then the record field must be declared in a record whose convention is CIL and the record must be declared in a package specification which is itself imported. In this case *String_Expression* must be a simple name (i.e. contains no dots) giving the name of the imported field.

4.2.4 Importing Dispatching Subprograms

If the *Local_Name* of an `Import` pragma is the name of a dispatching subprogram (i.e., a primitive operation of a tagged type), then the subprogram must be declared in a package specification which is itself imported. In this case *String_Expression* must be a simple name (i.e. contains no dots) giving the name of the imported subprogram.

4.2.5 Importing Objects

If the *Local_Name* of an `Import` pragma is the name of an object and the object is declared in a package specification which is itself imported, then the *String_Expression* must be a simple name (i.e. contains no dots) giving the name of the imported .NET static field.

An `Import` pragma for an object can be given even though such an entity does not occur in a package spec with an `Import` pragma. In this case the *String_Expression* of the `Import` pragma must give the complete .NET name of the imported entity, as shown in the following example:

```
procedure Foo is
   Var : Integer;
   pragma Import (CIL, Var, "pack.Foo.the_var");
begin
   Var := 3;
end Foo;
```

4.2.6 Importing Non-Dispatching Subprograms

If the *Local_Name* of an `Import` pragma is the name of a non-dispatching subprogram and the subprogram is declared in a package specification which is itself imported, then the *String_Expression* must be a simple name (i.e. contains no dots) giving the name of the imported .NET static method.

An `Import` pragma for a non-dispatching subprogram can be given even though such an entity does not occur in a package spec with an `Import` pragma. In this case the *String_Expression* of the `Import` pragma must give the complete .NET name of the imported entity as shown in the following example:

```
procedure Foo is
   X : Integer;
   function Compute (I : Integer) return Integer;
   pragma Import (CIL, Compute, "pack.Bar.calc");
begin
   X := Compute (3);
end Foo;
```

4.2.7 Importing Delegates

Starting with GNAT 6.2, access-to-subprograms Ada types and .NET delegates now perfectly match. As a result, an access-to-subprogram type can now be directly imported from a .NET delegate.

An `Import` pragma for an access-to-subprogram can be given even though such an entity does not occur in a package spec with an `Import` pragma. In this case the *String_Expression* of the `Import` pragma must give the complete .NET name of the imported entity as shown in the following example:

```
procedure Foo is
   --  This defines the delegate type, that matches pack.Bar.some_delegate_type
   type CB_Type is access procedure (Arg : Integer);
   pragma Import (CIL, CB, "pack.Bar.some_delegate_type");

   --  Let's import a method asking for such delegate as input.
   procedure Fn_Using_Delegate (CB : CB_Type);
   pragma Import (CIL, Fn_Using_Delegate, "pack.Bar.some_method");

   --  Our actual callback, full Ada
   procedure Bar (Arg : Integer);
begin
   --  We can now call the external .NET method with our full Ada Bar callback.
   Fn_Using_Delegate (Bar'Access);
end Foo;
```

Note that this behavior changed in GNAT 6.2. In previous versions, .NET delegates were treated as objects, and could only be imported as such.

4.3 Pragma Export CIL

In the absence of pragma `Export`, the name of any Ada object, field, or subprogram compiled into a class file is the name of the corresponding Ada entity folded to lower-case.

For exceptions, record types, and packages, the names of the generated class files are all folded to lower case.

By using pragma `Export` you can change the default name that is generated by the GNAT compiler. In addition, for Ada packages the pragma can also specify which .NET package they belong to. For convention CIL, the pragma `Export` has the following syntax:

```
pragma Export ([Convention    =>] CIL,
               [Entity        =>] Local_Name
             [,[External_Name =>] String_Expression]);
```

where *Local_Name* is the name of an object, subprogram, record component, record type, exception, or package, and *String_Expression* is a string giving the CIL name of the exported entity. If *String_Expression* is missing it is taken to be the *Local_Name*, folded to lower-case.

4.3.1 Exporting Objects, Subprograms, and Record Components

NOTE: Exporting of record components is not yet supported.

If the *Local_Name* of an `Export` pragma is the name of an object, record component, or subprogram (but not a top-level subprogram), *String_Expression* must be a simple name (i.e., it contains no dots), giving the name of the corresponding entity at the CIL level. Here is an example:

```
package C is
   type Typ is tagged record
      Field : Integer;
      pragma Export(CIL, Field, "THE_FIELD");
   end record;

   function Instance_Op (This : access Typ; I : Integer) return Integer;

   Var : Integer;
   function Op (J : Integer) return Integer;

private
   pragma Export (CIL, Instance_Op, "dispatch_op");
   pragma Export (CIL, Var, "the_var");
end C;
```

This is interpreted as the following two class specifications at the CIL level:

```
public class c {
   public static int the_var;
   public static int op (int j);
}
public class c$typ {
   public int THE_FIELD;
   public int dispatch_op (int i) {...}
}
```

Note that when exporting an object, subprogram, or record component you cannot specify its class, as this is determined by the compiler.

4.3.2 Exporting Exceptions

If the *Local_Name* of an `Export` pragma is the name of an exception *E*, then the *String_Expression* of such an `Export` pragma gives the name of the generated class for the Ada exception, overriding the name that would have been given by the compiler. *String_Expression* can be a simple class name, or it can have the form

```
namespace_name.class_name
```

indicating that the generated class belongs to .NET package *namespace_name*.

Care must be taken not to use the same class name for two Ada exceptions, packages or record types when they belong to different source files located in the same directory, since one '`.class`' file would overwrite the other.

4.3.3 Exporting Record Types

If the *Local_Name* of an `Export` pragma is the name of a record type *P*, then the *String_Expression* of such an `Export` pragma gives the name of the generated .NET class, overriding the name that would have been given by the compiler. *String_Expression* can be a simple class name, or it can have the form *namespace_name.class_name*.

Care must be taken not to use the same class name for two Ada exceptions or record types when they belong to different source files located in the same directory.

5 Debugging Ada Programs

Because GNAT generates DLLs and executables that are fully compliant with the .NET framework, you can use any .NET debugger (e.g. `Visual Studio`), with GNAT. However, in order to use such a debugger on Ada constructs that are not directly available in CIL (e.g. attributes), you need to know how GNAT compiles these into bytecode.

This chapter explains the correspondence between Ada features and bytecode. It is not a complete description; if you need to understand the output of the GNAT compiler for a particular Ada construct that is not documented below, you can inspect the assembly code generated by GNAT ('`.il`' file).

5.1 Ada Compilation Units

Unless pragma `Export` is used (see), the names of all classes generated from the compilation of an Ada unit are folded to lower case. Similarly for the names of all of the entities generated inside a class file.

The compilation of a nongeneric Ada library unit P always generates an assembly file '`p.il`' containing a `p_pkg` class.

A package `Q` nested inside an Ada unit `P` does not result in a separate class. Entities declared within the nested package will generally be associated as members of the containing library package's class. However, the names of the corresponding fields and methods resulting from the nested package will be given expanded names that include the name of the outermost library package followed by the names of any enclosing nested packages. Adjacent pairs of simple names in the expansion are separated by an underscore (e.g., '`p_q_proc`').

A child unit `P.Q` is compiled into a .NET class p.q_pkg. All the rules described here are applied recursively with respect to `Q`'s contents.

A generic package instantiation `R` nested inside an Ada unit `P` is treated exactly like a nested package.

A generic subprogram instantiation `S` nested inside an Ada unit `P` treated exactly like a nested subprogram.

A subprogram `N` nested inside another subprogram `P` will be treated as a static method of the enclosing library unit's class and will be given an expanded name that includes the names of any enclosing subprograms (e.g., '`pkg_p_n`'). In addition, a special class will be generated for the nested subprogram's enclosing subprogram to contain fields for any objects of the enclosing subprogram that are referenced by the nested subprogram. The name of this special Activation Record class is constructed by appending the prefix '`__AR_`' to the name of the enclosing subprogram (e.g., '`__AR_pkg_p`').

5.2 Lexical Elements

Letters in all Ada identifiers in user code are folded into lower case when generating symbolic references for .NET, unless `pragma Export` is used. However, certain names corresponding to internal entities generated by the GNAT front end may include upper-case letters. These can be seen using the `-gnatG` or `-gnatD` switches.

5.3 Enumeration Types

An Ada enumeration type is converted into a .NET 1-byte, 2-byte, 4-byte or 8-byte integer whose size best matches the value of the largest enumeration literal.

Character types are treated like regular Ada enumeration types. More specifically, the Ada `Character` type is mapped to a .NET `byte`, and the Ada `Wide_Character` type is mapped to the equivalent 2-byte .NET `char` type.

An Ada Boolean type is treated like a standard Ada enumeration type with 2 values and is consequently mapped into a .NET `byte`.

5.4 Integer Types

Each signed integer type is mapped to the smallest corresponding .NET integer type whose size is able to represent all required integer values:

Ada type	.NET type
Short_Short_Integer	byte *(1 byte)*
Short_Integer	short *(2 bytes)*
Integer	int *(4 bytes)*
Long_Integer	long *(8 bytes)*
Long_Long_Integer	long *(8 bytes)*

Each modular type is also mapped to the smallest corresponding .NET integer type whose size is able to represent all required modular values.

5.5 Floating Point Types

The Ada predefined floating point types map very naturally onto .NET's IEEE 32-bit float and IEEE 64-bit double:

Ada type	.NET type
Short_Float	float *(4 bytes)*
Float	float *(4 bytes)*
Long_Float	double *(8 bytes)*
Long_Long_Float	double *(8 bytes)*

User-defined floating point types are mapped into `Float` where possible, and `Long_Float` otherwise.

6 Limitations

Due to constraints of the .NET environment, or to implementation limitations, GNAT for .NET only supports a subset of the Ada language and GNAT run-time.

Language constructs not supported (where noted, partial support is provided):

- Types imported from .NET do not support enumeration attributes (e.g. `'Image`)
- Exception streams and attributes
- Representation items (13.1)
- pragma `Pack` (ignored) (13.2)
- Representation attributes (13.3)
- Record layout (13.5)
- Machine Code Insertions (13.8)
- `Unchecked_Conversion` between different non scalar types (13.9)
- Limited support on Ada.Streams package (13.13)
- `'Size` attribute on non scalar objects
- `'Storage_Size` attribute on non-task objects
- `'First_Bit`, `'Last_Bit`, `'Position` attributes
- `'External_Tag` attribute
- `'Pred`, `'Succ` attribute for modular types
- `'Version` and `'Body_Version` attributes
- Limited support of the `'Val` attribute
- Function returning unconstrained array will have wrong `'First` and `'Last`
- User-defined Storage_Pools
- Limited support for controlled types
- `'Address` on non-aliased, non-local objects
- `System.Address` comparisons, other than "=" and "/="
- Some forms of scalar object renaming (e.g. renaming of dereferenced access value)
- Pragmas `Import`, `Export`, and `Convention` other than `Ada` and `CIL`
- Pragma `Interrupt_Handler`, `Attach_Handler`
- Asynchronous abort of tasking constructs and tasks
- Access-to-protected-subprogram types
- Incomplete types completed in package bodies
- Stack overflows cannot be caught. (e.g ACATS test cb1010c & cb1010d) This limitation comes with .NET 2.0, which does not allow an application to catch the `StackOverflowException`.
- `Wide_String` and `Wide_Wide_String` (e.g. ACATS test c250001)
- Null arrays with multiple dimensions

Switches not supported:

- -gnatE (dynamic elaboration)

Run-time units not supported yet, which will be available in the future:

- Ada.Directories

Run-time units not supported:

- Ada.Sequential_IO.C_Streams, Ada.Storage_IO, Ada.Text_IO.C_Streams, Ada.Wide_Text_IO.C_Streams, Ada.Direct_IO.C_Streams
- Ada.Real_Time.Timing_Events
- Ada.Asynchronous_Task_Control
- Ada.Command_Line.Environment
- Ada.Exceptions.Traceback
- Ada.Interrupts,
- Ada.Task_Attributes,
- Ada.Task_Termination,
- Interfaces.C.Extensions, Interfaces.Cobol, Interfaces.C.Pointers, Interfaces.CPP, Interfaces.C.Strings, Interfaces.Fortran, Interfaces.Packed_Decimal
- Machine_Code, System.Machine_Code
- GNAT.Altivec
- GNAT.Array_Split
- GNAT.Lock_Files, GNAT.Socket
- GNAT.Exceptions, GNAT.Expect, GNAT.AWK, GNAT.CGI, GNAT.CRC32, GNAT.MD5, GNAT.SHA1, GNAT.Spitbol
- GNAT.Byte_Swapping
- GNAT.Calendar
- GNAT.Command_Line
- GNAT.Compiler_Version
- GNAT.Current_Exception, GNAT.Debug_Pools, GNAT.Debug_Utilities, GNAT.Exception_Actions, GNAT.Exception_Traces, GNAT.Memory_Dump
- GNAT.Dynamic_Tables
- GNAT.Float_Control
- GNAT.OS_Lib is only partially supported
- GNAT.Perfect_Hash_Generators
- GNAT.Secondary_Stack_Info
- GNAT.Table,
- GNAT.Task_Stack_Usage,
- GNAT.Time_Stamp
- GNAT.Thread, GNAT.Signal
- GNAT.String_Split, GNAT.Wide_String_Split, GNAT.Wide_Wide_String_Split GNAT.Traceback

Table of Contents