Ada Utility Library Programmer's Guide

STEPHANE CARREZ

2018-02-04

Contents

1 Introduction

The Ada Utility Library provides a collection of utility packages which includes:

- A logging framework close to Java log4j framework,
- A support for properties,
- A serialization/deserialization framework for XML, JSON, CSV,
- Ada beans framework,
- Encoding/decoding framework (Base16, Base64, SHA, HMAC-SHA, AES256),
- A composing stream framework (raw, files, buffers, pipes, sockets, compress),
- Several concurrency tools (reference counters, counters, pools, fifos, arrays),
- Process creation and pipes,
- Support for loading shared libraries (on Windows or Unix),
- HTTP client library on top of CURL or AWS.

This document describes how to build the library and how you can use the different features to simplify and help you in your Ada application.

2 Installation

This chapter explains how to build and install the library.

2.1 Before Building

Before building the library, you will need:

- XML/Ada
- AWS

First get, build and install the XML/Ada and then get, build and install the Ada Utility Library.

2.2 Configuration

The library uses the `configure` script to detect the build environment, check whether XML/Ada, AWS, Curl support are available and configure everything before building. If some component is missing, the `configure` script will report an error or it will disable the feature. The `configure` script provides several standard options and you may use:

- `--prefix=DIR` to control the installation directory,
- `--enable-shared` to enable the build of shared libraries,
- `--disable-static` to disable the build of static libraries,
- `--disable-traceback` to disable the support for symbolic traceback by the logging framework,
- `--disable-ahven` to disable building the Ahven support used by the Ada utility testing framework,
- `--enable-aunit` to enable building the AUnit support used by the Ada utility testing framework,
- `--disable-curl` to disable the support for CURL,
- `--disable-aws` to disable the support for AWS,
- `--with-xmlada=PATH` to control the installation path of XML/Ada,
- `--with-aws=PATH` to control the installation path of AWS,
- `--with-ada-lzma=PATH` to control the installation path of Ada LZMA,
- `--help` to get a detailed list of supported options.

In most cases you will configure with the following command:

```
1 ./configure
```

2.3 Build

After configuration is successful, you can build the library by running:

```
1 make
```

After building, it is good practice to run the unit tests before installing the library. The unit tests are built and executed using:

```
1 make test
```

And unit tests are executed by running the `bin/util_harness` test program.

2.4 Installation

The installation is done by running the `install` target:

```
1 make install
```

If you want to install on a specific place, you can change the `prefix` and indicate the installation direction as follows:

```
1 make install prefix=/opt
```

3 Logging

The `Util.Log` package and children provide a simple logging framework inspired from the Java Log4j library. It is intended to provide a subset of logging features available in other languages, be flexible, extensible, small and efficient. Having log messages in large applications is very helpful to understand, track and fix complex issues, some of them being related to configuration issues or interaction with other systems. The overhead of calling a log operation is negligeable when the log is disabled as it is in the order of 30ns and reasonable for a file appender has it is in the order of 5us.

3.1 Using the log framework

A bit of terminology:

- A *logger* is the abstraction that provides operations to emit a message. The message is composed of a text, optional formatting parameters, a log level and a timestamp.
- A *formatter* is the abstraction that takes the information about the log to format the final message.
- An *appender* is the abstraction that writes the message either to a console, a file or some other final mechanism.

3.2 Logger Declaration

Similar to other logging framework such as Java Log4j and Log4cxx, it is necessary to have and instance of a logger to write a log message. The logger instance holds the configuration for the log to enable, disable and control the format and the appender that will receive the message. The logger instance is associated with a name that is used for the configuration. A good practice is to declare a `Log` instance in the package body or the package private part to make available the log instance to all the package operations. The instance is created by using the `Create` function. The name used for the configuration is free but using the full package name is helpful to control precisely the logs.

```
1 with Util.Log.Loggers;
2 package body X.Y is
3   Log : constant Util.Log.Loggers := Util.Log.Loggers.Create ("X.Y");
4 end X.Y;
```

3.3 Logger Messages

A log message is associated with a log level which is used by the logger instance to decide to emit or drop the log message. To keep the logging API simple and make it easily usable in the application,

several operations are provided to write a message with different log level.

A log message is a string that contains optional formatting markers that follow more or less the Java `MessageFormat` class. A parameter is represented by a number enclosed by `{}`. The first parameter is represented by `{0}`, the second by `{1}` and so on. Parameters are replaced in the final message only when the message is enabled by the log configuration. The use of parameters allows to avoid formatting the log message when the log is not used.

The example below shows several calls to emit a log message with different levels:

```
1  Log.Error ("Cannot open file {0}: {1}", Path, "File does not exist");
2  Log.Warn ("The file {0} is empty", Path);
3  Log.Info ("Opening file {0}", Path);
4  Log.Debug ("Reading line {0}", Line);
```

The logger also provides a special `Error` procedure that accepts an Ada exception occurence as parameter. The exception name and message are printed together with the error message. It is also possible to activate a complete traceback of the exception and report it in the error message. With this mechanism, an exception can be handled and reported easily:

```
1  begin
2     ...
3  exception
4     when E : others =>
5        Log.Error ("Something bad occurred", E, Trace => True);
6  end;
```

3.4 Log Configuration

The log configuration uses property files close to the Apache Log4j and to the Apache Log4cxx configuration files. The configuration file contains several parts to configure the logging framework:

- First, the *appender* configuration indicates the appender that exists and can receive a log message.
- Second, a root configuration allows to control the default behavior of the logging framework. The root configuration controls the default log level as well as the appenders that can be used.
- Last, a logger configuration is defined to control the logging level more precisely for each logger.

Here is a simple log configuration that creates a file appender where log messages are written. The file appender is given the name `result` and is configured to write the messages in the file `my-log-file.log`. The file appender will use the `level-message` format for the layout of messages. Last is the configuration of the `X.Y` logger that will enable only messages starting from the `WARN` level.

```
log4j.rootCategory=DEBUG,result
log4j.appender.result=File
log4j.appender.result.File=my-log-file.log
log4j.appender.result.layout=level-message
log4j.logger.X.Y=WARN
```

By default when the `layout` is not set or has an invalid value, the full message is reported and the generated log messages will look as follows:

```
[2018-02-07 20:39:51] ERROR - X.Y - Cannot open file test.txt: File
   does not exist
[2018-02-07 20:39:51] WARN  - X.Y - The file test.txt is empty
[2018-02-07 20:39:51] INFO  - X.Y - Opening file test.txt
[2018-02-07 20:39:51] DEBUG - X.Y - Reading line ......
```

When the `layout` configuration is set to `data-level-message`, the message is printed with the date and message level.

```
[2018-02-07 20:39:51] ERROR: Cannot open file test.txt: File does not
   exist
[2018-02-07 20:39:51] WARN : The file test.txt is empty
[2018-02-07 20:39:51] INFO : X.Y - Opening file test.txt
[2018-02-07 20:39:51] DEBUG: X.Y - Reading line ......
```

When the `layout` configuration is set to `level-message`, only the message and its level are reported.

```
ERROR: Cannot open file test.txt: File does not exist
WARN : The file test.txt is empty
INFO : X.Y - Opening file test.txt
DEBUG: X.Y - Reading line ......
```

The last possible configuration for `layout` is `message` which only prints the message.

```
Cannot open file test.txt: File does not exist
The file test.txt is empty
Opening file test.txt
Reading line ......
```

4 Property Files

The `Util.Properties` package and children implements support to read, write and use property files either in the Java property file format or the Windows INI configuration file. Each property is assigned a key and a value. The list of properties are stored in the `Util.Properties.Manager` tagged record and they are indexed by the key name. A property is therefore unique in the list. Properties can be grouped together in sub-properties so that a key can represent another list of properties.

4.1 File formats

The property file consists of a simple name and value pair separated by the = sign. Thanks to the Windows INI file format, list of properties can be grouped together in sections by using the `[section-name]` notation.

```
test.count=20
test.repeat=5
[FileTest]
test.count=5
test.repeat=2
```

4.2 Using property files

An instance of the `Util.Properties.Manager` tagged record must be declared and it provides various operations that can be used. When created, the property manager is empty. One way to fill it is by using the `Load_Properties` procedure to read the property file. Another way is by using the `Set` procedure to insert or change a property by giving its name and its value.

In this example, the property file `test.properties` is loaded and assuming that it contains the above configuration example, the `Get ("test.count")` will return the string `"20"`. The property `test.repeat` is then modified to have the value `"23"` and the properties are then saved in the file.

```
with Util.Properties;
...
   Props : Util.Properties.Manager;
   ...
      Props.Load_Properties (Path => "test.properties");
      Ada.Text_IO.Put_Line ("Count: " & Props.Get ("test.count");
      Props.Set ("test.repeat", "23");
      Props.Save_Properties (Path => "test.properties");
```

To be able to access a section from the property manager, it is necessary to retrieve it by using the `Get` function and giving the section name. For example, to retrieve the `test.count` property of the `FileTest` section, the following code is used:

```
1    FileTest : Util.Properties.Manager := Props.Get ("FileTest");
2    ...
3       Ada.Text_IO.Put_Line ("[FileTest] Count: "
4                             & FileTest.Get ("test.count");
```

When getting or removing a property, the `NO_PROPERTY` exception is raised if the property name was not found in the map. To avoid that exception, it is possible to check whether the name is known by using the `Exists` function.

```
1    if Props.Exists ("test.old_count") then
2       ... --  Property exist
3    end if;
```

4.3 Reading JSON property files

The `Util.Properties.JSON` package provides operations to read a JSON content and put the result in a property manager. The JSON content is flattened into a set of name/value pairs. The JSON structure is reflected in the name. Example:

```
1 { "id": "1",                                  id         -> 1
2   "info": { "name": "search",                 info.name  -> search
3             "count", "12",                    info.count -> 12
4             "data": { "value": "empty" }},    info.data.value  -> empty
5   "count": 1                                  info.count -> 1
6 }
```

To get the value of a JSON property, the user can use the flatten name. For example:

```
1  Value : constant String := Props.Get ("info.data.value");
```

The default separator to construct a flatten name is the dot (`.`) but this can be changed easily when loading the JSON file by specifying the desired separator:

```
1  Util.Properties.JSON.Read_JSON (Props, "config.json", "|");
```

Then, the property will be fetch by using:

```
1  Value : constant String := Props.Get ("info|data|value");
```

4.4 Property bundles

Property bundles represent several property files that share some overriding rules and capabilities. Their introduction comes from Java resource bundles which allow to localize easily some configuration files or some message. When loading a property bundle a locale is defined to specify the target language and locale. If a specific property file for that locale exists, it is used first. Otherwise, the property bundle will use the default property file.

A rule exists on the name of the specific property locale file: it must start with the bundle name followed by _ and the name of the locale. The default property file must be the bundle name. For example, the bundle `dates` is associated with the following property files:

```
1 dates.properties           Default values (English locale)
2 dates_fr.properties        French locale
3 dates_de.properties        German locale
4 dates_es.properties        Spain locale
```

Because a bundle can be associated with one or several property files, a specific loader is used. The loader instance must be declared and configured to indicate one or several search directories that contain property files.

```
1 with Util.Properties.Bundles;
2 ...
3    Loader : Util.Properties.Bundles.Loader;
4    Bundle : Util.Properties.Bundles.Manager;
5    ...
6    Util.Properties.Bundles.Initialize (Loader,
7                                        "bundles;/usr/share/bundles");
8    Util.Properties.Bundles.Load_Bundle (Loader, "dates", "fr", Bundle);
9    Ada.Text_IO.Put_Line (Bundle.Get ("util.month1.long");
```

In this example, the `util.month1.long` key is first searched in the `dates_fr` French locale and if it is not found it is searched in the default locale.

The restriction when using bundles is that they don't allow changing any value and the `NOT_WRITEABLE` exception is raised when one of the `Set` operation is used.

When a bundle cannot be loaded, the `NO_BUNDLE` exception is raised by the `Load_Bundle` operation.

4.5 Advance usage of properties

The property manager holds the name and value pairs by using an Ada Bean object.

It is possible to iterate over the properties by using the `Iterate` procedure that accepts as parameter a `Process` procedure that gets the property name as well as the property value. The value itself is passed as an `Util.Beans.Objects.Object` type.

5 Date Utilities

The `Util.Dates` package provides various date utilities to help in formatting and parsing dates in various standard formats. It completes the standard `Ada.Calendar.Formatting` and other packages by implementing specific formatting and parsing.

5.1 Date Operations

Several operations allow to compute from a given date:

- `Get_Day_Start`: The start of the day (0:00),
- `Get_Day_End`: The end of the day (23:59:59),
- `Get_Week_Start`: The start of the week,
- `Get_Week_End`: The end of the week,
- `Get_Month_Start`: The start of the month,
- `Get_Month_End`: The end of the month

The `Date_Record` type represents a date in a split format allowing to access easily the day, month, hour and other information.

```
1 Now        : Ada.Calendar.Time := Ada.Calendar.Clock;
2 Week_Start : Ada.Calendar.Time := Get_Week_Start (Now);
3 Week_End   : Ada.Calendar.Time := Get_Week_End (Now);
```

5.2 RFC7231 Dates

The RFC7231 defines a standard date format that is used by HTTP headers. The `Util.Dates.RFC7231` package provides an `Image` function to convert a date into that target format and a `Value` function to parse such format string and return the date.

```
1   Now  : constant Ada.Calendar.Time := Ada.Calendar.Clock;
2   S    : constant String := Util.Dates.RFC7231.Image (Now);
3   Date : Ada.Calendar.time := Util.Dates.RFC7231.Value (S);
```

A `Constraint_Error` exception is raised when the date string is not in the correct format.

5.3 ISO8601 Dates

The ISO8601 defines a standard date format that is commonly used and easily parsed by programs. The `Util.Dates.ISO8601` package provides an `Image` function to convert a date into that target format

and a `Value` function to parse such format string and return the date.

```
1   Now  : constant Ada.Calendar.Time := Ada.Calendar.Clock;
2   S    : constant String := Util.Dates.ISO8601.Image (Now);
3   Date : Ada.Calendar.time := Util.Dates.ISO8601.Value (S);
```

A `Constraint_Error` exception is raised when the date string is not in the correct format.

5.4 Localized date formatting

The `Util.Dates.Formats` provides a date formatting operation similar to the Unix `strftime` or the `GNAT.Calendar.Time_IO`. The localization of month and day labels is however handled through `Util.Properties.Bundle` (similar to the Java world). Unlike `strftime`, this allows to have a multi-threaded application that reports dates in several languages. The `GNAT.Calendar.Time_IO` only supports English and this is the reason why it is not used here.

The date pattern recognizes the following formats:

Format	Description
%a	The abbreviated weekday name according to the current locale.
%A	The full weekday name according to the current locale.
%b	The abbreviated month name according to the current locale.
%h	Equivalent to %b. (SU)
%B	The full month name according to the current locale.
%c	The preferred date and time representation for the current locale.
%C	The century number (year/100) as a 2-digit integer. (SU)
%d	The day of the month as a decimal number (range 01 to 31).
%D	Equivalent to %m/%d/%y
%e	Like %d, the day of the month as a decimal number,
	but a leading zero is replaced by a space. (SU)
%F	Equivalent to %Y-%m-%d (the ISO 8601 date format). (C99)
%G	The ISO 8601 week-based year
%H	The hour as a decimal number using a 24-hour clock (range 00 to 23).
%I	The hour as a decimal number using a 12-hour clock (range 01 to 12).

Format	Description
%j	The day of the year as a decimal number (range 001 to 366).
%k	The hour (24 hour clock) as a decimal number (range 0 to 23);
%l	The hour (12 hour clock) as a decimal number (range 1 to 12);
%m	The month as a decimal number (range 01 to 12).
%M	The minute as a decimal number (range 00 to 59).
%n	A newline character. (SU)
%p	Either "AM" or "PM"
%P	Like %p but in lowercase: "am" or "pm"
%r	The time in a.m. or p.m. notation.
	In the POSIX locale this is equivalent to %I:%M:%S %p. (SU)
%R	The time in 24 hour notation (%H:%M).
%s	The number of seconds since the Epoch, that is,
	since 1970-01-01 00:00:00 UTC. (TZ)
%S	The second as a decimal number (range 00 to 60).
%t	A tab character. (SU)
%T	The time in 24 hour notation (%H:%M:%S). (SU)
%u	The day of the week as a decimal, range 1 to 7,
	Monday being 1. See also %w. (SU)
%U	The week number of the current year as a decimal
	number, range 00 to 53
%V	The ISO 8601 week number
%w	The day of the week as a decimal, range 0 to 6,
	Sunday being 0. See also %u.
%W	The week number of the current year as a decimal number,
	range 00 to 53
%x	The preferred date representation for the current locale
	without the time.

Format	Description
%X	The preferred time representation for the current locale without the date.
%y	The year as a decimal number without a century (range 00 to 99).
%Y	The year as a decimal number including the century.
%z	The timezone as hour offset from GMT.
%Z	The timezone or name or abbreviation.

The following strftime flags are ignored:

Format	Description
%E	Modifier: use alternative format, see below. (SU)
%O	Modifier: use alternative format, see below. (SU)

SU: Single Unix Specification C99: C99 standard, POSIX.1-2001

See strftime (3) manual page

To format and use the localize date, it is first necessary to get a bundle for the `dates` so that date elements are translated into the given locale.

```
1  Factory     : Util.Properties.Bundles.Loader;
2  Bundle      : Util.Properties.Bundles.Manager;
3  ...
4     Load_Bundle (Factory, "dates", "fr", Bundle);
```

The date is formatted according to the pattern string described above. The bundle is used by the formatter to use the day and month names in the expected locale.

```
1  Date : String := Util.Dates.Formats.Format (Pattern => Pattern,
2                                              Date    => Ada.Calendar.
                                                  Clock,
3                                              Bundle  => Bundle);
```

6 Ada Beans

A [Bean|Java] is an object that allows to access its properties through getters and setters. Java Beans rely on the use of Java introspection to discover the Java Bean object properties.

An Ada Bean has some similarities with the Java Bean as it tries to expose an object through a set of common interfaces. Since Ada does not have introspection, some developer work is necessary. The Ada Bean framework consists of:

- An `Object` concrete type that allows to hold any data type such as boolean, integer, floats, strings, dates and Ada bean objects.
- A `Bean` interface that exposes a `Get_Value` and `Set_Value` operation through which the object properties can be obtained and modified.
- A `Method_Bean` interface that exposes a set of method bindings that gives access to the methods provided by the Ada Bean object.

The benefit of Ada beans comes when you need to get a value or invoke a method on an object but you don't know at compile time the object or method. That step being done later through some external configuration or presentation file.

The Ada Bean framework is the basis for the implementation of Ada Server Faces and Ada EL. It allows the presentation layer to access information provided by Ada beans.

6.1 Objects

The `Util.Beans.Objects` package provides a data type to manage entities of different types by using the same abstraction. The `Object` type allows to hold various values of different types.

An `Object` can hold one of the following values:

- a boolean,
- a long long integer,
- a date,
- a string,
- a wide wide string,
- a generic data

Several operations are provided to convert a value into an `Object`.

```
1 Value : Util.Beans.Objects.Object := Util.Beans.Objects.To_Object ("
     something");
2 Value := Value + To_Object ("12");
```

6.2 Datasets

The `Datasets` package implements the `Dataset` list bean which defines a set of objects organized in rows and columns. The `Dataset` implements the `List_Bean` interface and allows to iterate over its rows. Each row defines a `Bean` instance and allows to access each column value. Each column is associated with a unique name. The row `Bean` allows to get or set the column by using the column name.

6.3 Bean Interface

An Ada Bean is an object which implements the `Util.Beans.Basic.Readonly_Bean` or the `Util.Beans.Basic.Bean` interface. By implementing these interface, the object provides a behavior that is close to the Java Beans: a getter and a setter operation are available.

7 HTTP

The `Util.Http` package provides a set of APIs that allows applications to use the HTTP protocol. It defines a common interface on top of CURL and AWS so that it is possible to use one of these two libraries in a transparent manner.

7.1 Client

The Util.Http.Clients package defines a set of API for an HTTP client to send requests to an HTTP server.

7.1.1 GET request

To retrieve a content using the HTTP GET operation, a client instance must be created. The response is returned in a specific object that must therefore be declared:

```
1 Http     : Util.Http.Clients.Client;
2 Response : Util.Http.Clients.Response;
```

Before invoking the GET operation, the client can setup a number of HTTP headers.

```
1 Http.Add_Header ("X-Requested-By", "wget");
```

The GET operation is performed when the Get procedure is called:

```
1 Http.Get ("http://www.google.com", Response);
```

Once the response is received, the Response object contains the status of the HTTP response, the HTTP reply headers and the body. A response header can be obtained by using the Get_Header function and the body using Get_Body:

```
1 Body : constant String := Response.Get_Body;
```

8 Streams

The `Util.Streams` package provides several types and operations to allow the composition of input and output streams. Input streams can be chained together so that they traverse the different stream objects when the data is read from them. Similarly, output streams can be chained and the data that is written will traverse the different streams from the first one up to the last one in the chain. During such traversal, the stream object is able to bufferize the data or make transformations on the data.

The `Input_Stream` interface represents the stream to read data. It only provides a `Read` procedure. The `Output_Stream` interface represents the stream to write data. It provides a `Write`, `Flush` and `Close` operation.

8.1 Buffered Streams

The `Output_Buffer_Stream` and `Input_Buffer_Stream` implement an output and input stream respectively which manages an output or input buffer. The data is first written to the buffer and when the buffer is full or flushed, it gets written to the target output stream.

The `Output_Buffer_Stream` must be initialized to indicate the buffer size as well as the target output stream onto which the data will be flushed. For example, a pipe stream could be created and configured to use the buffer as follows:

```
1  with Util.Streams.Buffered;
2  with Util.Streams.Pipes;
3  ...
4     Pipe   : aliased Util.Streams.Pipes.Pipe_Stream;
5     Buffer : Util.Streams.Buffered.Output_Buffer_Stream;
6     ...
7        Buffer.Initialize (Output => Pipe'Unchecked_Access,
8                           Size => 1024);
```

In this example, the buffer of 1024 bytes is configured to flush its content to the pipe input stream so that what is written to the buffer will be received as input by the program. The `Output_Buffer_Stream` provides write operation that deal only with binary data (`Stream_Element`). To write text, it is best to use the `Print_Stream` type from the `Util.Streams.Texts` package as it extends the `Output_Buffer_Stream` and provides several operations to write character and strings.

The `Input_Buffer_Stream` must also be initialized to also indicate the buffer size and either an input stream or an input content. When configured, the input stream is used to fill the input stream buffer. The buffer configuration is very similar as the output stream:

```
with Util.Streams.Buffered;
with Util.Streams.Pipes;
...
   Pipe   : aliased Util.Streams.Pipes.Pipe_Stream;
   Buffer : Util.Streams.Buffered.Input_Buffer_Stream;
   ...
      Buffer.Initialize (Input => Pipe'Unchecked_Access, Size => 1024);
```

In this case, the buffer of 1024 bytes is filled by reading the pipe stream, and thus getting the program's output.

8.2 Texts

The Util.Streams.Texts package implements text oriented input and output streams. The Print_Stream type extends the Output_Buffer_Stream to allow writing text content.

The Reader_Stream package extends the Input_Buffer_Stream and allows to read text content.

8.3 File streams

The Util.Streams.Files package provides input and output streams that access files on top of the Ada Stream_IO standard package.

8.4 Pipes

The `Util.Streams.Pipes` package defines a pipe stream to or from a process. It allows to launch an external program while getting the program standard output or providing the program standard input. The `Pipe_Stream` type represents the input or output stream for the external program. This is a portable interface that works on Unix and Windows.

The process is created and launched by the `Open` operation. The pipe allows to read or write to the process through the `Read` and `Write` operation. It is very close to the *popen* operation provided by the C stdio library. First, create the pipe instance:

```
with Util.Streams.Pipes;
...
   Pipe : aliased Util.Streams.Pipes.Pipe_Stream;
```

The pipe instance can be associated with only one process at a time. The process is launched by using the `Open` command and by specifying the command to execute as well as the pipe redirection mode:

- `READ` to read the process standard output,
- `WRITE` to write the process standard input.

For example to run the `ls -l` command and read its output, we could run it by using:

```
1  Pipe.Open (Command => "ls -l", Mode => Util.Processes.READ);
```

The `Pipe_Stream` is not buffered and a buffer can be configured easily by using the `Input_Buffer_Stream` type and connecting the buffer to the pipe so that it reads the pipe to fill the buffer. The initialization of the buffer is the following:

```
1  with Util.Streams.Buffered;
2  ...
3     Buffer : Util.Streams.Buffered.Input_Buffer_Stream;
4     ...
5     Buffer.Initialize (Input => Pipe'Unchecked_Access, Size => 1024);
```

And to read the process output, one can use the following:

```
1  Content : Ada.Strings.Unbounded.Unbounded_String;
2  ...
3  Buffer.Read (Into => Content);
```

The pipe object should be closed when reading or writing to it is finished. By closing the pipe, the caller will wait for the termination of the process. The process exit status can be obtained by using the `Get_Exit_Status` function.

```
1  Pipe.Close;
2  if Pipe.Get_Exit_Status /= 0 then
3     Ada.Text_IO.Put_Line ("Command exited with status "
4                           & Integer'Image (Pipe.Get_Exit_Status));
5  end if;
```

You will note that the `Pipe_Stream` is a limited type and thus cannot be copied. When leaving the scope of the `Pipe_Stream` instance, the application will wait for the process to terminate.

Before opening the pipe, it is possible to have some control on the process that will be created to configure:

- The shell that will be used to launch the process,
- The process working directory,
- Redirect the process output to a file,
- Redirect the process error to a file,

- Redirect the process input from a file.

All these operations must be made before calling the `Open` procedure.

8.5 Sockets

The Util.Streams.Sockets package defines a socket stream.

8.6 Raw files

The Util.Streams.Raw package provides a stream directly on top of file system operations read and write.

8.7 Encoding Streams

The Encoding_Stream tagged record represents a stream with encoding capabilities. The stream passes the data to be written to the Transformer interface that allows to make transformations on the data before being written.

```
1  Encode : Util.Streams.Buffered.Encoders.Encoding_Stream;
```

The encoding stream manages a buffer that is used to hold the encoded data before it is written to the target stream. The Initialize procedure must be called to indicate the target stream, the size of the buffer and the encoding format to be used.

```
1  Encode.Initialize (Output => File'Access, Size => 4096, Format => "
      base64");
```

8.7.1 Encoder and Decoders

The Util.Encoders package defines the Encoder and Decode objects which provide a mechanism to transform a stream from one format into another format.

Simple encoding and decoding

8.8 Timer Management

The Util.Events.Timers package provides a timer list that allows to have operations called on regular basis when a deadline has expired. It is very close to the Ada.Real_Time.Timing_Events package but it provides more flexibility by allowing to have several timer lists that run independently. Unlike the GNAT implementation, this timer list management does not use tasks at all. The timer list can therefore be used in a mono-task environment by the main process task. Furthermore you can control your own task priority by having your own task that uses the timer list.

The timer list is created by an instance of Timer_List:

```
1 Manager : Util.Events.Timers.Timer_List;
```

The timer list is protected against concurrent accesses so that timing events can be setup by a task but the timer handler is executed by another task.

8.8.1 Timer Creation

A timer handler is defined by implementing the Timer interface with the Time_Handler procedure. A typical timer handler could be declared as follows:

```
1 type Timeout is new Timer with null record;
2 overriding procedure Time_Handler (T : in out Timeout);
3 My_Timeout : aliased Timeout;
```

The timer instance is represented by the Timer_Ref type that describes the handler to be called as well as the deadline time. The timer instance is initialized as follows:

```
1 T : Util.Events.Timers.Timer_Ref;
2 Manager.Set_Timer (T, My_Timeout'Access, Ada.Real_Time.Seconds (1));
```

8.8.2 Timer Main Loop

Because the implementation does not impose any execution model, the timer management must be called regularly by some application's main loop. The Process procedure executes the timer handler that have ellapsed and it returns the deadline to wait for the next timer to execute.

```
1 Deadline : Ada.Real_Time.Time;
2 loop
3    ...
```

```
4     Manager.Process (Deadline);
5     delay until Deadline;
6  end loop;
```

9 Performance Measurements

Performance measurements is often made using profiling tools such as GNU gprof or others. This profiling is however not always appropriate for production or release delivery. The mechanism presented here is a lightweight performance measurement that can be used in production systems.

The Ada package `Util.Measures` defines the types and operations to make performance measurements. It is designed to be used for production and multi-threaded environments.

9.1 Create the measure set

Measures are collected in a `Measure_Set`. Each measure has a name, a counter and a sum of time spent for all the measure. The measure set should be declared as some global variable. The implementation is thread safe meaning that a measure set can be used by several threads at the same time. It can also be associated with a per-thread data (or task attribute).

To declare the measure set, use:

```
1  with Util.Measures;
2     ...
3     Perf : Util.Measures.Measure_Set;
```

9.2 Measure the implementation

A measure is made by creating a variable of type `Stamp`. The declaration of this variable marks the begining of the measure. The measure ends at the next call to the `Report` procedure.

```
1  with Util.Measures;
2  ...
3    declare
4       Start : Util.Measures.Stamp;
5    begin
6       ...
7       Util.Measures.Report (Perf, Start, "Measure for a block");
8    end;
```

When the `Report` procedure is called, the time that elapsed between the creation of the `Start` variable and the procedure call is computed. This time is then associated with the measure title and the associated counter is incremented. The precision of the measured time depends on the system. On GNU/Linux, it uses `gettimeofday`.

If the block code is executed several times, the measure set will report the number of times it was executed.

9.3 Reporting results

After measures are collected, the results can be saved in a file or in an output stream. When saving the measures, the measure set is cleared.

```
1  Util.Measures.Write (Perf, "Title of measures",
2                       Ada.Text_IO.Standard_Output);
```

9.4 Measure Overhead

The overhead introduced by the measurement is quite small as it does not exceeds 1.5 us on a 2.6 Ghz Core Quad.

9.5 What must be measured

Defining a lot of measurements for a production system is in general not very useful. Measurements should be relatively high level measurements. For example:

- Loading or saving a file
- Rendering a page in a web application
- Executing a database query