ZanyBlue Ada Libraries and Applications

Michael Rohan

Version: 1.3.0 Beta, r2990X

Contents

Chapter 1

Introduction

This manual describes the use of the ZanyBlue libraries, an Ada framework for finite element analysis with supporting libraries which might be of general use in other projects. The initial implementation contains the `Text` package (and supporting packages and utilities) supplying functionality mirroring the Java `MessageFormat` package. The `Text` functionality allows the application messages to be externalized in `.properties` files and localized into additional languages. As a major side effect the formatting of arguments within messages is needed and implemented.

This, naturally, led to a need for a testing framework. AUnit is used for unit testing of the libraries, however, testing of the command line utilities proved more difficult. While DejaGNU was available, it seemed a little more involved than was needed for this project. A new testing application was written, `zbtest`, which is a hierarchical testing framework supporting definition scopes.

This document is available as a single PDF file../zanyblue.pdf.

Package documentation is available via GNAT HTMLref/index.html generated file.

1.1 Building the Library and Applications

Building the ZanyBlue library and applications does not require a configure step (at this time). The build requirements are similar on Unix (Linux) and Windows (the Makefiles use the environment variable `OS` to determine whether the build is Linux (the variable is not defined) or Windows (where it is defined with the value `Windows_NT`).

1.1.1 Requirements

The build requires

- A recent version of GNAT (e.g., GPL 2010, 2011) from AdaCore.
- GNU make.
- The XMLAda package needs to be installed to build the ZanyBlue testing application, `zbtest`.
- The unit tests require the AUnit package, e.g., GPL 2011 version.

1.1.2 Building

Once the required environment is in place, building is simply a matter of issuing "`make`" in the "`src`" directory, e.g.,

```
$ make -C src
```

This produces the library in the "`lib/zanyblue`" directory, and the two executables "`zbmcompile`", the message compiler, and "`zbtest`" the ZanyBlue testing application. The ZanyBlue package spec files and generic bodies are copied to the "`include/zanyblue`" directory.

1.1.3 Using with GNAT

A ZanyBlue GNAT `gprbuild` project file is available in the directory "`lib/gnat`" directory. Adding this directory to your "`ADA_PROJECT_PATH`" should allow the use of the ZanyBlue library with `gprbuild` projects. See the text examples and the GPS build instructions.

1.1.4 Testing

Testing is via the "`check`" target. From the "`src`" directory this will run both the unit and system tests. If experimenting, running these tests independently is probably more useful (the combined testing is normally only run from under Jenkins where the summary XML files are loaded as the test results for the build):

```
$ make -C src/test/unittest check
$ make -C src/test/system check
```

1.1.5 Examples

The examples include a Gtk example. If Gtk is not installed, this example will fail to build (but the other should be OK). The "`dumplocale`" example is useful when examining the built-in locale data, e.g., to see the date, time, numeric, etc data for Swedish,

```
$ make -C examples/text/dumplocale
$ make -C examples/text/dumplocale run_sv
```

1.1.6 Windows Issues

The installation of the Windows GNU Win32 version of make, http://gnuwin32.sourceforge.net/package does not update the Path environment. This should be done manually after installing via the Control Panel → System → Advanced → Environment Variables → Path and adding the path `C:\Program Files\GnuWin32\bin`.

The majority of the example application are text based. This is problematic on Windows systems as the standard Windows command console does not understand Unicode output. This makes the Gtk example application the only fully functional example application on Windows systems. To build this application, the GtkAda package should be installed, again from AdaCore and, as for AUnit, the gprbuild project path environment variable will need to be set if GtkAda is not installed in the default location.

Chapter 2

The Root Package

The ZanyBlue root package, `ZanyBlue` contains definitions about the library, primiarly version information. The version numbers associated with the library are available via the functions

- `Version_Major`, the major version number for the ZanyBlue release.
- `Version_Minor`, the minor version number for the ZanyBlue release.
- `Version_Patch`, the patch number for the ZanyBlue release.

Each release also has a release status given by the function

```
function Version_Status_Type return Version_Status_Type;
```

The return value is an enumeration type: Alpha, Beta, Production.

To futher refine the release identification, a source code revision value (a wide string) is returned by the function `Revision`. The source code system is currently Subversion and this string has the format of the letter "`r`" followed by the Subversion revision number, e.g., "`r2500`".

Finally, the copyright year for the bundle is given by the function

```
function Copyright_Year return Positive;
```

This is current year at the time of the build.

Chapter 3

The Text Package

The intent of the `Text` package is to allow the separate definition of the text messages displayed by an application into a `.properties` file. Messages are reference in the source code by a key value (a simple string or accessor function returning the text associated with a particular key) within a group of related messages—a facility. The message strings include Java style message argument place holders: argument number (zero-based) along with optional type specific formatting information enclosed in chain brackets. Allowing formatting of message arguments in the context of a localized message.

The divorce of the message text from the source code allows the easy localization of application messages. The source language `.properties` file is delivered to translation vendors who then return localized `.properties` files. E.g., for an application `myapp`, with application messages defined in the file

```
myapp.properties
```

the localized French and Japanese files would be

```
myapp_fr.properties
myapp_ja.properties
```

The `Text` package supports message text searching among a set of localized `.properties` files with fallback to available messages. For example, if the above application, `myapp`, is run in a Canadian French environment (`ZB_LANG=fr_CA`), localized display of each message is implemented by trying to locate the message in the files, in order,

```
myapp_fr_CA.properties
myapp_fr.properties
myapp.properties
```

If no French localized message is available, i.e., the properties file which does not include a language code is used. Frequently this is English.

3.1 Messages and Arguments

The localization of a message cannot, in general, occur using just the text 'snippets' surrounding values generated by an application, e.g., the externally visible message

```
There are 2 moons orbiting "Mars".
```

could be divided into the 'snippets'

- "`There are`"
- "`moons orbiting`"
- "`.`"

concatenated together at runtime along with formatted values for the number of moons and the planet name. If delivered to a translation vendor, the 'snippet' would needs to localize independently. This is a rather difficult task to do for general text 'snippets' as the order of the 'snippets' is defined by the application and cannot be changed by localization vendors. The 'snippets' themselves are generally too short to give translators sufficient context. As a general rule, translation should be performed on complete sentences.

A complete sentence is possible if place holders are used for message arguments. Using Java style arguments, the above message would be defined as

```
There are {0} moons orbiting "{1}".
```

The application message arguments are referenced within chain brackets. In the example here, argument 0 would be the integer `2` and argument 1 would be the string"`Mars`".

This message is then stored in a `.properties` file, external to the application and given a key name, e.g., to use they key string "`moons`" to refer to the message in the application code, the `.properties` file definition would be

```
moons=There are {0} moons orbiting "{1}".
```

This sentence can be localized independently of the application and, since the English string is also defined externally to the application, this text can also be changed. For example, if a technical writer review of the English used in an application decided, for consistency reasons, to rephrase the sentence, just the English properties file need be adjusted, e.g.,

```
moons=The planet "{1}" has {0} moons.
```

This model of searching `.properties` files is a conceptual model in the context of the ZanyBlue Text library. The ZanyBlue implementation compiles the set of properties files for a facility into Ada code and localized message resolution for a particular key occurs against run-time data structures rather than accessing files. This makes for an efficient implementation where the major portion of the data processing occurs at application build time. See details on the `zbmcompile` (**??**), the ZanyBlue message compiler, later.

3.2 Example Using Low Level Functions

As an introduction to the `Text` package, a simple example, which continues the example based on the number of moons orbiting the various planets, is developed here.

Note, the preferred way to access messages is using accessor packages described in the next section. The example here uses direct calls using strings to identify facilities and key to demonstrate the concepts involved.

The task is to ask the user for a planet name and print the number of currently known moons for the planet. The source for this simple example is available in the directory `examples/text/moons` and is listed here:

```
with Ada.Wide_Text_IO;
with Moons_Messages;
with ZanyBlue.Text.Formatting;

procedure Moons is

   type Planet_Names is (Mercury, Venus, Earth, Mars,
                         Jupiter, Saturn, Uranus, Neptune);
   package Planet_Name_IO is
      new Ada.Wide_Text_IO.Enumeration_IO (Planet_Names);

   use Ada.Wide_Text_IO;
   use Planet_Name_Formatting;
   use ZanyBlue.Text.Formatting;
```

```
   Moons  : array (Planet_Names) of Natural := (
                  Earth => 1, Mars => 2, Jupiter => 63,
                  Saturn => 62, Uranus => 27, Neptune => 13,
                  others => 0);
   Planet : Planet_Names;

begin
   loop
      Print ("moons", "0001");
      Planet_Name_IO.Get (Planet);
      if Moons (Planet) /= 1 then
         Print_Line ("moons", "0002", +Moons (Planet),
                                      +Planet_Names'Image (Planet));
      else
         Print_Line ("moons", "0003", +Planet_Names'Image (Planet));
      end if;
   end loop;
exception
when End_Error | Data_Error =>
    New_Line;
    Print_Line ("moons", "0004");
end Moons;
```

The example source code uses the `Generic_Enumerations` ZanyBlue package for planet name arguments. Generic ZanyBlue packages should be instanciated at the library level.

Here the messages for the application are referred to by message id or key, i.e., numeric strings, e.g., "`0001`" for the facility "`moons`".[1] The text associated with these message keys are externalized to a `.properties` file. For this example, the root properties file, "`moons.properties`," containing English, is

```
0001=Please enter a planet:
0002=There are {0} known moons orbiting "{1}".
0003=There is 1 known moon orbiting "{0}".
0004=OK, goodbye.
```

A German translation of this properties, "`moons_de.properties`", file would be (via Google Translate)

```
0001=Bitte geben Sie einen Planeten:
0002=Es gibt {0} bekannte Monde umkreisen "{1}".
0003=Es gibt 1 bekannt Mond umkreisen "{0}".
0004=OK, auf Wiedersehen.
```

[1] As a side note, the key is a general case sensitive string, however, the assignment of more meaningful names becomes more difficult as an application grows and it is simpler to abandon such names initially and use numeric string codes.

French and Spanish Google translated version of this file in the examples directory.

The properties and application are tied together by "compiling" the properties files into an Ada package and simply **with**'ing the compiled package in the application (normally in the unit containing the main procedure). In this example, the generated package referenced in the source above is `Moons_Messages` which is created using the `zbmcompile` ZanyBlue message compiler utility:

```
$ zbmcompile -i -v Moons_Messages moons
This is ZBMCompile, Version 1.0.0 BETA (r2621) at 8:25:33 AM on Feb 23, 2012
Copyright (c) 2009-2012, Michael Rohan.  All rights reserved
Loaded 16 messages for the facility "moons" (4 locales)
Performing consistency checks for the facility "moons"
Loaded 1 facilities, 4 keys, 4 locales and 16 messages
Loaded total 517 characters, stored 517 unique characters, 0% saving
Wrote the spec "Moons_Messages" to the file "./moons_messages.ads"
Wrote the body "Moons_Messages" to the file "./moons_messages.adb"
ZBMCompile completed at 8:25:33 AM on Feb 23, 2012, elapsed time 0:00:00.057
```

This generates an Ada specification containing the name of the facility, "moons" in this case, and the definition of a single initialization routine. The body file contains the message text data, the initialization routine which adds the message data to a shared message pool. Since the `-i` command line option was used, the generated package body includes a call to the initialization routine allowing the message data to be included in the application via a simple **with** of the package, i.e., no explicit call to the initialization routine is needed.

The execution of the generated application will now display messages selected for the run-time locale, e.g., in an English locale:

```
$ make run
../../../bin/x_moons
Please enter a planet: mars
There are 2 known moons orbiting "MARS".
Please enter a planet: earth
There is 1 known moon orbiting "EARTH".
Please enter a planet: mercury
There are 0 known moons orbiting "MERCURY".
Please enter a planet: ^D
OK, goodbye.
```

And running the application selecting a German locale displays the German messages:

```
$ make run_de
../../../bin/x_moons de
Bitte geben Sie einen Planeten: mars
```

```
Es gibt 2 bekannte Monde umkreisen "MARS".
Bitte geben Sie einen Planeten: earth
Es gibt 1 bekannt Mond umkreisen "EARTH".
Bitte geben Sie einen Planeten: mercury
Es gibt 0 bekannte Monde umkreisen "MERCURY".
Bitte geben Sie einen Planeten: ^D
OK, auf Wiedersehen.
```

The locale is selected via a command line argument here, however, the environment variables can be used instead, see section **??** for more details on selecting locales.

This example application did not attempt to localize the planet names. A more involved example would add another facility for the planet names with message keys matching the `Image` of the enumeration values.

3.3 Example Using Accessor Functions

The style of message generation using a facility string, key string and message arguments opens an application up to the risk of a mis-match between the expected arguments for a message and the supplied arguments (possibly causing the ZanyBlue exceptions `No_Such_Argument_Error` or `No_Such_Key` to be raised). It is also possible to silently, in the sense the compiler does not compile, introduce errors by accidentally mis-spelling a facility or key value (again possibly causing the exceptions `No_Such_Facility_Error` `No_Such_Key_Error` to raised).

For an Ada application, where the compile time detection of errors is an expectation, this is not ideal. To overcome this, the ZanyBlue message compiler utility can generate accessor packages for the compiled facilities. These packages define functions and procedures for each key in the facility with a list of arguments matching the expected number of arguments. This allows the Ada compiler to perform checks on the references to messages of ZanyBlue-ised applications.

The `zbmcompile` utility generates three major classes of accessor packages (when the `-a` option is given):

1. Accessor functions which return the localized formatted message for a message key. There are two accessor function packages generated:
   - A package for functions with return typed of `Wide_String`.
   - A package for functions with return typed of `String` (the standard Wide Strings are encoding using the encoding schema defined by the locale, the default being UTF-8 encoding).

In both cases, the functions generated are named by using message key prefixed with the string "`Format_`". This requires message keys, when prefixed with "`Format_`" be valid Ada identifier.

2. Accessor that print the localized formatted message similar to the standard `Put_Line` routines. There are two styles of routines generated:

   - Routines that use the standard `Ada.Wide_Text_IO` routines to print the formatted messages.
   - Routines that use the standard `Ada.Text_IO` routines to print the formatted messages as encoded strings using the encoding for the select locale.

3. Accessor procedures which raise an exception with a localized formatted message (similar to the standard `Raise_Exception` procedure but extended to allow message arguments). The procedures names generated prefix the message key with the string "`Raise_`".

These accessor packages are generated for each facility compiled and are created as child packages of the command line package name.

As an example, reworking the moons example in terms of the safer accessor functions, the code would be, e.g., for the message that prints the number of moons for a planet, using the generated wide print routine:

```
with Messages.Moons_Wide_Prints;
...
Print_0002 (+Moons (Planet), +Planet));
```

Here, the compiled messages are generated to the `Messages` package and the accessor packages are child packages of it.

For this accessor example, the `zbmcompile` utility must be used with the "`-a`" command line option to generate the accessor packages and the target parent package is given as "`Messages`":

```
$ zbmcompile -a -i -v Messages Moons
This is ZBMCompile, Version 0.2.0 ALPHA (r2621) at 8:58:04 AM on Feb 21, 2012
Copyright (c) 2009-2012, Michael Rohan.  All rights reserved
Loaded 18 messages for the facility "Moons" (4 locales)
Performing consistency checks for the facility "Moons"
Performing consistency checks for the accessor package generation
Loaded 1 facilities, 5 keys, 4 locales and 18 messages
Loaded total 551 characters, stored 551 unique characters, 0% saving
Wrote the spec "Messages" to the file "./messages.ads"
Wrote the body "Messages" to the file "./messages.adb"
Generated accessor package spec "Messages.Moons_Exceptions"
    to "./messages-moons_exceptions.ads"
```

```
Generated accessor package body "Messages.Moons_Exceptions"
    to "./messages-moons_exceptions.adb"
Generated accessor package spec "Messages.Moons_Strings"
    to "./messages-moons_strings.ads"
Generated accessor package body "Messages.Moons_Strings"
    to "./messages-moons_strings.adb"
Generated accessor package spec "Messages.Moons_Wide_Strings"
    to "./messages-moons_wide_strings.ads"
Generated accessor package body "Messages.Moons_Wide_Strings"
    to "./messages-moons_wide_strings.adb"
Generated accessor package spec "Messages.Moons_Prints"
    to "./messages-moons_prints.ads"
Generated accessor package body "Messages.Moons_Prints"
    to "./messages-moons_prints.adb"
Generated accessor package spec "Messages.Moons_Wide_Prints"
    to "./messages-moons_wide_prints.ads"
Generated accessor package body "Messages.Moons_Wide_Prints"
    to "./messages-moons_wide_prints.adb"
ZBMCompile completed at 8:58:04 AM on Feb 21, 2012, elapsed time 0:00:00.176
```

(Long output lines have been wrapped).

The minor difference in the naming of the properties files should also be noted: for the example in the previous section, the files, and hence the facility, were all lower case. Here, the base file name is "Moons". This allow the generated Ada packages to have the more expected name "`Moons_Wide_Strings`".

Since the messages are now, in a sense, handled by the Ada compiler, IDE's, e.g., GPS, will display the message text (the generated packages include the message text, with markers explained later). The accessor display in GPS for the example message above is given in figure **??**. The keys used for the messages in this example are simple numeric style strings. The GPS IDE will also display the base language text and the number of arguments expected for accessors as the generated Ada code includes this text as a comment.

3.4 Message Formatting

From the simple "moons" example above, it can be seen that simply externalizing just the text used to in an application without the argument formatting is not enough to fully support localization. The messages externalized must be the complete messages, i.e., sentences, with embedded place holders for the arguments substituted at runtime.

The ZanyBlue Text library currently uses a mixture of Java and Python

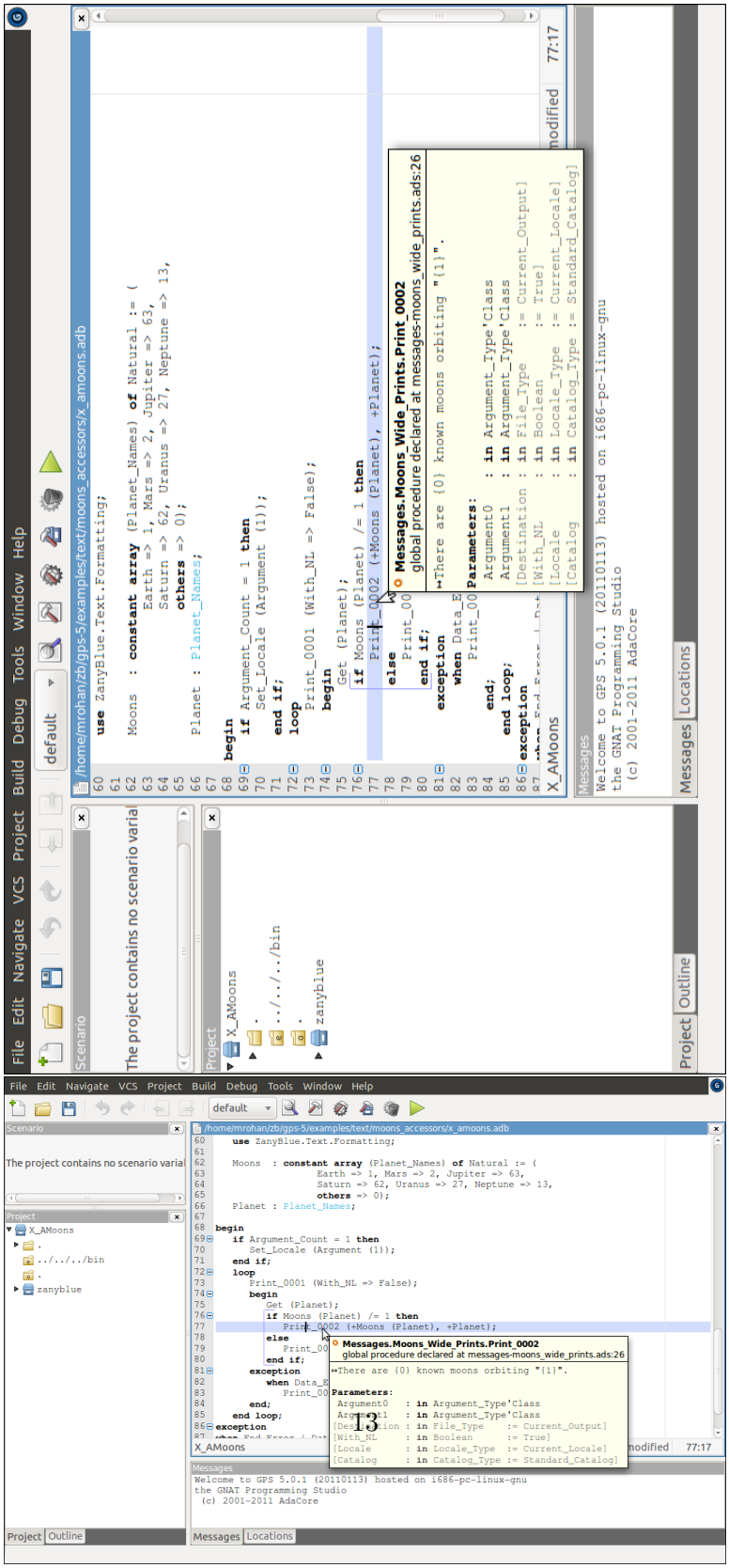


Figure 3.1: GPS display for an accessor function.

styles for embedded arguments. Arguments to the message are referenced by index (zero based) and enclosed in chain brackets.

From the "moons" example, message "0002" has the definition

```
0002=There are {0} known moons orbiting "{1}".
```

Here the first argument (argument 0) is an integer and second argument (argument 1) is an enumeration value giving the planet name.

At runtime, arguments to messages are "boxed" into a tagged type with dispatching methods that perform the formatting to strings. Each type has it's own implementation (see "Provided Argument Types" later, **??**). Boxing occurs by creating a boxed object for the argument value and passed to the various message formatting or printing routines. The boxing function has a standard "`Create`" function and a renaming "`+`" making formatting or printing calls look more natural. For example, the message text above could be formatted with an integer argument, 2, and a string argument, "Mars", as

```
Message : Wide_String := Format ("moons", "0002", +2, +String'("Mars"));
```

or, if the accessor package is used,

```
Message : Wide_String := Format_0002 (+2, +String'("Mars"));
```

The explicit type coercion to `String` is required as both standard `String` and `Wide_String` are supported.

The formatting or printing routines accept up to 5 optional boxed arguments. The implementation gathers the supplied boxed arguments into an argument list and then calls the underlying formatting routine with the argument list. For argument numbers beyond 5, the underlying argument list type must be used and the arguments must be explicitly appended, e.g., the above example could be rephrased in terms of the lower level argument list based formatting routine as

```
declare
   Arguments : Argument_List;
begin
   Arguments.Append (1);
   Arguments.Append (String'("Mars"));
   Display_Message (Format ("moons", "0002", Arguments));
end;
```

This 5 argument limit does *not* apply to the functions and procedures generated for accessor packages. These routines have arguments lists that match the number of expected arguments without the 5 argument limit.

Type Name	Description
`any`	No specific type required.
`boolean`	Boolean values required.
`character`	Character (wide or narrow) values required.
`date`	A Calendar type required (formatted as a date).
`datetime`	A Calendar type required (formatted as a date and time).
`duration`	Duration type required.
`enum`	Enumeration type required.
`exception`	Exception type required (e.g., `when E : others`)
`fixed`	A fixed point value is required.
`float`	A floating point value is required.
`integer`	An integer value is required.
`modular`	A modular type value is required.
`number`	A numeric value (integer, real, etc) is required.
`real`	A real value (float or fixed) is required.
`string`	A string (wide or narrow) is required (characters also acceptable).
`time`	A Calendar type required (formatted as a time).

Table 3.1: Format type names.

3.4.1 Formatting Syntax

Additional formatting information can be included in the argument reference via an optional format string after the index value, separated by either a comma or a colon. The format syntax is based on the syntax defined for Python format strings in most cases (see the Argument Types section for details on types that use additional, non-Python style formatting, e.g, dates and time).

Specifying Argument Types

The format string is optionally prefixed by a type name separated by a comma character for the expected argument without space character (which would be interpreted as part for the format). E.g., to indicate a particular argument is should be an integer, the format would be

```
0002=There are {0,integer} known moons orbiting "{1}".
```

The type names recognized are given in table **??**.

The type information, apart from the date and time related names, do not impact runtime formatting (the boxed value is formatted according to

the boxed type formatting routine). The type information does, however, impact the signature of accessor routine generated.

For example, the message,

```
0002=There are {0,integer} known moons orbiting "{1}".
```

would generate a format style access with the signature

```
function Format_0002 (
   Argument0   : Integer_Category_Type'Class;
   Argument1   : Any_Category_Type'Class;
   Locale      : Locale_Type  := Current_Locale;
   Catalog     : Catalog_Type := Standard_Catalog) return Wide_String;
```

Here, arugment 0 is required to be an integer type (verified by the compiler) while argument 1 is unconstrainted with respect to type.

Use of the accessor allows the compiler to verify arguments have the expected type.

Type names have the expected hierarchy, e.g., a numeric argument type allows integer, float, fixed, etc, arguments.

General Formatting Syntax

The formatting information for the various types supported, in general, follow the Python/C style embedded formatting format for all but the date and time related formatting (the formatting of these values is described later in section **??**). E.g., to format an integer with a field width of 10 characters, the argument reference would be

```
0001=Total number is {0,integer,10}
```

The syntax for the format information is (using standard BNF notation)

```
[[fill]align][sign][#][0][width][.precision][type][qual]
```

where

fill The fill character to use when padding, e.g., if the formatted value string is less than the field width additional padding characters are added fill out the field. Where the characters are added is defined by the next character, `align`. The fill character cannot be the '}' character.

align The align character defines the alignment of the formatted value within the minimum field width. The alignment characters recognized are

'<' Value is flushed left, any additional padding characters needed are added to the right of the value.

'>' Value is flushed right, any additional padding characters needed are prepend to the left of the value.

'=' Any padding characters needed are added after the sign character, e.g., '+00010'. This is used only with numeric value. For non-numeric values, this alignment character is interpreted as '<'.

'^' The value is centered in the field width with any padding character being added before and after the value to center it.

Note that unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so the alignment option has no meaning in this case.

sign The sign character defines how the sign of numeric values are handled. This applies only to the sign of the value, the exponent of floating point numbers formatted in scientific notation is always preceeded by a sign character. It has three valid values:

'+' A sign character is always generated. Positive values are preceeded by the locale plus character, negative values by the locale minus character.

'-' A sign character is only generated for negative values where the locale minus character is used. This is the default behaviour if no sign character is specified.

' ' A sign character is only generated for negative values where the locale minus character is used, positive values are preceeded by a space character.

Note, the sign format character is ignored for non-numeric arguments.

The hash character causes integer formatted values formatted as binary, octal or hexadecimal to be decorated with the base using standard Ada notation. E.g., formatting the integer 2012 using base 16:

"x" gives 7dc

"#x" gives 16#7dc#

0 If the width field is preceded by a zero ('0') character, this enables zero-padding. This is equivalent to an alignment type of '=' and a fill character of '0'.

width The width is a Latin integer value defining the minimum field width for the argument. Padding, using the fill character, is added if needed to meet this minimum width.

precision The precision is a Latin integer value indicating how many digits should be displayed after the decimal point for a floating point value. The precision is not used for non-floating type formatting.

type The type character gives the expected data type of the numeric argument. It is not an error if the argument is not numeric (type specific accessors can be used to ensure type compatibility). The integer type characters supported are

'b' Binary format. Outputs the number in base 2.

'd' Decimal Integer. Outputs the number in base 10.

'o' Octal format. Outputs the number in base 8.

'x' Hex format. Outputs the number in base 16, using lowercase letters for the digits above 9.

'X' Hex format. Outputs the number in base 16, using uppercase letters for the digits above 9.

None The same as 'd'.

The floating point type characters supported are:

'E' Exponent notation. Prints the number in scientific notation using the localized equivalent of the letter 'E' to indicate the exponent.

'e' Exponent notation. Same as 'E'. Other formatting systems, e.g., C, would use case difference in the format string to change the case of the exponent character in the formatted value. Since localzied versions are being used, it is not clear if lowercasing/uppercasing such strings is valid. The two format characters are treated the same.

'F' Fixed point. Displays the number as a fixed-point number.

'f' Fixed point. Same as 'F'.

'G' General format. This prints the number as a fixed-point number, unless the number is too large, in which case it switches to 'E' exponent notation. Infinity and NaN values are formatted as using localized versions.

'g' General format. Same as 'G'.

None The same as 'E'.

qual The format string can be terminated with a final qualifier character. For the current version, the only valid value for this character is '*' which forces the formatting of the value using the Root locale, i.e., standard Ada Latin formatting. This impact the formatting of date and time values and the formatting of numbers where a localized version might use localized digits instead of the Latin "0123456789", e.g., Arabic locales.

3.5 Argument Types

The formatting of messages with arguments is based on "boxing" message argument data values. The library provides a set of standard boxed types corresponding to the standard set of Ada types. Details on how the formatting these types is detailed in the following sections.

3.5.1 Boolean Type

Localized version of the English strings "true" and "false" are not available via the Unicode CLDR data. Boolean values are formatted as unlocalized English values. Examples, for format references are (the '*' is used to fill for readability):

```
Format ("{0}", +True)       →  "true"
Format ("{0,*>10}", +True)  →  "******true"
Format ("{0,*<10}", +True)  →  "true******"
Format ("{0,*^10}", +True)  →  "***true***"
```

Other formatting information is ignored, e.g., a base character associated with numeric types, etc. The accessor argument type for Booleans is `Boolean_Category_Type` which is derived from the `Enumeration_Category_Type`.

3.5.2 Character Type

The character implementation simply inserts the character value as is to the formatted output. The library support both narrow and wide characters. Since "Create" exist for both, type information must be supplied if literal values are used. Field width, fill and alignment are respected, e.g., for the character variable 'C' with a value of 'a':

```
Format ("{0}", +C)       →  "a"
Format ("{0,*>10}", +C)  →  "*********a"
Format ("{0,*<10}", +C)  →  "a*********"
Format ("{0,*^10}", +C)  →  "*****a****"
```

Other formatting information is ignored, e.g., the type specifiers. The accessor argument type for characters is `Character_Category_Type` which is derived from the `String_Category_Type`, i.e., a character is considered a "degenerate" string.

3.5.3 Duration Type

The standard implementation of the duration formatting displays the days, hours, minutes and seconds (seconds as a floating point formatted to three decimal places). If the number days is zero, it's formatting is suppressed. The digits used are localized (this only impacts Arabic locales). Field width, fill and alignment are respected, e.g., for the Duration variable '`D`'

```
Format ("{0}", +D)       →  "16:48:03.000"
Format ("{0,*>16}", +D)  →  "****16:48:03.000"
Format ("{0,*<16}", +D)  →  "16:48:03.000****"
Format ("{0,*^16}", +D)  →  "**16:48:03.000**"
```

The accessor argument type for Durations is `Duration_Category_Type`.

3.5.4 Float Types

The `Float` and `Long_Float` formatting are simply instantiations of the `Generic_Floats` package, see section **??** for more information. Some examples, details explained in the description of the generic package:

```
Format ("{0,e}", +Pi);    →  "3.14159E+00"
Format ("{0,f}", +Pi);    →  "3.141590"
Format ("{0,g}", +Pi);    →  "3.141590"
Format ("{0,.2f}", +Pi);  →  "3.14"
```

The accessor argument type for floats is `Float_Category_Type` which is derived from the `Real_Category_Type`.

3.5.5 Enumeration Types

Formatting of Ada enumeration types is handled via the generic package `Generic_Enumerations` which should be instanciated using the enumeration type. This allows enumeration values as arguments to messages. Obviously, no localization is available for the enumeration values (generated using

`Wide_Image`). The generic package supplies a `Create` function to create the ZanyBlue "boxed" value along with a renaming for "`+`".

Field width, fill and alignment are respected.

For example, for the definitions

```
type Colour is (Red, Green, Blue);
package Colour_Arguments is
   new ZanyBlue.Text.Generic_Enumerations (Colour);
use Colour_Arguments;
C : Colour := Red;
```

the example formatting is

```
Format ("{0}", +C);        →  "RED"
Format ("{0,*<10}", +C);   →  "*******RED"
Format ("{0,*>10}", +C);   →  "RED*******"
Format ("{0,*^10}", +C);   →  "****RED***"
```

The underlying implementation uses the `Image` function for the enumeration value and, as a result, is normally an uppercase value. The format width and placement (center, left, right) is respected. The accessor argument type for enumerations is `Enumeration_Category_Type`.

If localization of the enumeration image is required, the application will need to implement this. One possibility would be to create another facility that uses the enumeration image as a key and returns the localized value, e.g., for the `Colour` example above,

```
RED=Red
GREEN=Green
BLUE=Blue
```

and then use this facility to generate arguments, e.g.,

```
Format ("The colour is {0}",
        +Format ("colours", Colours'Wide_Image (C)));
```

There is, of course, a lot more work if the application were to support input of localized enumeration names.

Generic ZanyBlue text packages should be instanciated at the library level to prevent run time accessibility exceptions.

3.5.6 Generic Float Types

The `Generic_Floats` package implements formatting for floating point type. The formatting is based on David M. Gay's [**?**] algorithm and attempts to

produce accruate representations of floating point numbers (see also Guy Steele and Jon White's paper [**?**]).

The formatting numeric style is controlled by the type characters (unlike C, the case of the character does not matter):

'`E`' The floating point number is formatted using scientific notation, e.g., `1.23E+10`. Note in addition to the digits, the decimal point, sign characters and the exponent character are localized, e.g., in a Swedish locale the exponent characters is displayed as "times 10 to the power of".

'`F`' The floating point number is formatted as a simple number. Note, for large absolute values of the exponent the formatted value will be a very long string of mainly zero characters.

'`G`' This format chooses the shorter of the '`E`' end '`F`' formatting depending on the value being formatted. For this release, the algorithm used is relative simplistic.

In additionl to the type characters, the formatting width and precision are used when formatting floating point numbers:

width The total field width the formatted value should occupy. If the formatted value is smaller than this width, the result is padded to fill to this width (see the alignment characters later).

precision The precision is the number of digit displayed after the decimal point.

The plus or space character can be used to force either a plus or space character before the formatted number for positive values, negative numbers always include a sign character. The sign characters used are locale defined.

The alignment and fill characters are used to pad the result to the requested field width. In addition to the left, right and center alignment, floating point (and numeric values in general) also support the numeric alignment character '`=`' which is simply a shorthand for align right using the '`0`' character for fill (this is the localized '`0`' character).

As an example, the following table gives the various formatting options for the `Float` value `F := 2.9979E+8`:

```
Format ("{0,e}", +F);       →  "2.99790E+08"
Format ("{0,f}", +F);       →  "299790000.0"
Format ("{0,g}", +F);       →  "2.99790E+08"
Format ("{0,.2f}", +F);     →  "299790000.00"
Format ("{0,.2e}", +F);     →  "3.00E+08"
Format ("{0,*>16e}", +F);   →  "*****2.99790E+08"
Format ("{0,=16e}", +F);    →  "000002.99790E+08"
```

The implementation will use localized strings for infinity and "not a number" when formatting such values.

The accessor argument type for floats is `Float_Category_Type` which is derived from the `Real_Category_Type`.

Generic ZanyBlue text packages should be instanciated at the library level to prevent run time accessibility exceptions.

3.5.7 Generic Integer Types

The `Generic_Integer` package implements formatting for integer types (generic argument type `range <>`).

The formatting numeric style is controlled by the type characters which control the base used when formatting, whether the value has a base decorator, handling of the sign, etc. The base selector characters are

'`b`' Format the value in binary (base 2).

'`o`' Format the value in octal (base 8).

'`d`' Format the value in decimal (base 10), this is the default.

'`x`' Format the value in hexadecinmal (base 16), extra digits are the lower case Latin characters '`a`' to '`f`'. The CLDR data does not supply localized hexadecimal digits.

'`X`' Format the value in hexadecinmal (base 16), extra digits are the upper case Latin characters '`A`' to '`F`'.

If the format string includes the base decorator character '`#`' then the non-decimal format include the base, as per Ada syntax in the formattted result.

The alignment and fill characters are used to pad the result to the requested field width. In addition to the left, right and center alignment, integer (and numeric values in general) also support the numeric alignment

character '=' which is simply a shorthand for align right using the '0' character for fill (this is the localized '0' character).

As an example, the following table gives the various formatting options for the `Integer` value `I := 42`:

```
Format ("{0}", +I);         →  "42"
Format ("{0,b}", +I);       →  "101010"
Format ("{0,o}", +I);       →  "52"
Format ("{0,x}", +I);       →  "2a"
Format ("{0,X}", +I);       →  "2A"
Format ("{0,#b}", +I);      →  "2#101010#"
Format ("{0,#o}", +I);      →  "8#52#"
Format ("{0,#x}", +I);      →  "16#2a#"
Format ("{0,#X}", +I);      →  "16#2A#"
Format ("{0,=#10X}", +I);   →  "16#00002A#"
```

The accessor argument type for integers is `Integer_Category_Type` which is derived from the `Number_Category_Type`.

Generic ZanyBlue text packages should be instanciated at the library level to prevent run time accessibility exceptions.

3.5.8 Generic Modular Types

The `Generic_Modulars` package implements formatting for modular types (generic type argument `mod <>`). This is essentially the same as the generic integers: the same rules apply, see section **??**.

The accessor argument type for integers is `Modular_Category_Type` which is derived from the `Integer_Category_Type`.

Generic ZanyBlue text packages should be instanciated at the library level to prevent run time accessibility exceptions.

3.5.9 Integer Type

This is simply an instantiation of the `Generic_Integers` package for the standard `Integer` type. See the generic integers description in section **??**.

3.5.10 String Types

The string implementation simply inserts the string value as is to the formatted output. The library support both narrow and wide fixed and unbounded strings. Since "`Create`" exist for both, type information must be supplied

if literal values are used. Field width, fill and alignment are respected, e.g., for the string variable '`S`' with a value of "`abc`":

```
Format ("{0}", +S)       →  "abc"
Format ("{0,*>10}", +S)  →  "*******abc"
Format ("{0,*<10}", +S)  →  "abc*******"
Format ("{0,*^10}", +S)  →  "****abc***"
```

Other formatting information is ignored, e.g., the type specifiers. The accessor argument type for characters is `String_Category_Type`.

3.5.11 Date and Time Types

The implementation for the formatting of times is the more involved and support two sub-categories to select either the time or date value of an `Ada.Calendar.Time` value. This is currently the only argument type that does not use the standard format string, e.g., you cannot specify a width, precision, etc. The root locale formatting is available, however, using a trailing '`*`' character in the format.

The built-in localization support includes localized formats for dates, times and date/times. These localization are implemented in terms of ASCII date/time format strings, e.g., the occurence of the sequence "`dd`" within the format generated the day of the month in the output to as two characters (0 padded). The full set of format strings is documented in table **??**. (Note, some locale can have more than the simple am, noon, and pm for the day period, see the `dumplocale` example.) Characters not part of a recognized format substring are simply copied to the output as is. Sub-strings that should be included as is can be enclosed in single quotes (this is only needed if the sub-string would otherwise be interpreted as date/time values.

The date and time formatting is localized using the information from the CLDR data and include localized day and month names along with localized date and time formats.

The day in the week from a day is calculated using the code from [**?**].

While a date/time format string can be included as part of the argument description, it is more normal to use the "pre-package", locale aware, format styles:

- '`short`', the basic information, e.g., a time format would likely not include the seconds. This is the default format.
- '`medium`', additional formatting on '`short`', e.g., a time format would likely include the seconds, abbreviated month names in dates, etc.

`a`	Day period name, e.g., `am`, `noon` or `pm`.
`d`	Day of the month, 1 ... 31
`dd`	Day of the month, 01 ... 31
`EEEE`	Full day of the week name
`EEE`	Abbreviated day of the week name
`G`	Era (CE/BCE, only CE is available with Ada)
`h`	Hours, 0 ... 12
`HH`	Hours, 00 ... 23
`H`	Hours, 0 ... 23
`mm`	Minutes, 00 ... 59
`m`	Minutes, 0 ... 59
`MMMM`	Full month name
`MMM`	Abreviated month name
`MM`	Month number 00 ... 12
`M`	Month number 0 ... 12
`ss`	Seconds, 00 ... 59
`s`	Seconds, 0 ... 59
`yyyy`	Full year, 2012, four digits
`yy`	Year, e.g., 12
`y`	Year, e.g., 2012, minimum number of digits
`zzzz`	Timezone names (not available, GMT offset printed)
`z`	GMT timezone offset printed

Table 3.2: The set of recognized format strings for date and times

- '`long`', the additional formatting on '`medium`', e.g., full month names, etc.
- '`full`', the additional formatting on '`long`', e.g., full day names, etc.

Using direct templates rather than the format styles might not work in all locales, i.e., generate mixed language results. E.g., if a template references the abbreviated month name, this will display as English in an Arabic locale (abbreviated month names do not appear to be used in Arabic).

The accessor argument type for characters is `Calendar_Category_Type`, all three format type strings ('`date`', '`time`' and '`datetime`' map to this category type.

There is a lot of variety with date and times so the examples are more extensive that other types. In the following, the date/time being formatted is 4:48 pm, Blooms Day, June 16, 1904, referred to via the variable '`B`'. The examples below use two locales for demonstration purposes: "`en_US`" and "`fr`".

Short date and times styles: Default

The simplest use is to not specify anything. The generates an accessor using an "`Any`" type category, in this case the '`datetime`' is used to format. A type name is encouraged as the compiler will type check message arguments.

```
Format ("{0}", +B)            →  en_US:  "4:48 PM 6/16/04"
                                    fr:  "16:48 16/06/04"
Format ("{0,time}", +B)       →  en_US   "4:48 PM"
                                    fr:  "16:48"
Format ("{0,date}", +B)       →  en_US:  "6/16/04"
                                    fr:  "16/06/04"
Format ("{0,datetime}", +B)   →  en_US:  "4:48 PM 6/16/04"
                                    fr:  "16:48 16/06/04"
```

The formatting when no style is specified is "`short`", e.g., `"{0,time}"` is the same as `"{0,time,short}"`.

Medium date and times styles

The medium style add more formatted values, e.g., using the example date. Due to space constraints, the formatting specifier is given rather than the full call to the `Format` function, e.g., `Format ("{0,time,long}", +B);` is written as `time,long`.

```
time,medium      →  en_US:  "4:48:03 PM"
                       fr:  "16:48:03"
date,medium      →  en_US:  "Jun 16, 1904"
                       fr:  "16 juin 1904"
datetime,medium  →  en_US:  "4:48:03 PM Jun 16, 1904"
                       fr:  "16:48:03 16 juin 1904"
```

Long date and times styles

The long style adds more formatted values on the medium style, e.g., again using the example date (example executed in the Pacific time zone, 7 hours earlier than GMT):

```
time,long      →  en_US:  "4:48:03 PM -0700"
                     fr:  "16:48:03 -0700"
date,long      →  en_US:  "June 16, 1904"
                     fr:  "16 juin 1904"
datetime,long  →  en_US:  "4:48:03 PM -0700 June 16, 1904"
                     fr:  "16:48:03 -0700 16 juin 1904"
```

Full date and times styles

Finally, the full style uses full day and month names, e.g., using the example date (example executed in the Pacific time zone, 7 hours earlier than GMT):

```
time,full      →  en_US:  "4:48:03 PM -0700"
                     fr:  "16:48:03 -0700"
date,full      →  en_US:  "Thursday, June 16, 1904"
                     fr:  "jeudi 16 juin 1904"
datetime,full  →  en_US:  "4:48:03 PM -0700 Thursday, June 16, 1904"
                     fr:  "16:48:03 -0700 jeudi 16 juin 1904"
```

3.6 Message Filtering

Frequently an application has different output modes, e.g., debug, verbose, normal, quiet, etc. If the various `Print` routines are used to generate messages for the application, these messages can be filtered using the tagged type `Message_Filter_Type` which is used by the ZanyBlue library to determine if a message should be really be printed.

The `Message_Filter_Type` defines the method

```
function Is_Filtered (Filter   : Message_Filter_Type;
                      Facility : Wide_String;
                      Key : Wide_String) return Boolean;
```

An application convention can be used on the message keys to define, e.g., verbose message filtering. The example application `examples/text/filtering` uses the convention:

- Verbose messages begin with the letter '`V`'.
- Informational messages begin with the letter '`I`'.
- Warning messages begin with the letter '`W`'.
- Error messages begin with the letter '`E`'.

The declaration of a filtering type for this configuration would be

```
type My_Filter_Type is new Message_Filter_Type with
   record
      Verbose : Boolean := False;
   end record;

function Is_Filtered (Filter   : My_Filter_Type;
                      Facility : Wide_String;
                      Key : Wide_String) return Boolean;
```

with the simple implementation for this example begin

```
function Is_Filtered (Filter   : My_Filter_Type;
                      Facility : Wide_String;
                      Key : Wide_String) return Boolean is
begin
   return Key (Key'First) = 'V' and not Filter.Verbose;
end Is_Filtered;
```

To enable the filtering, the filter must be registered with the ZanyBlue library via the `Set_Filter` routine. This routine takes an access to `Message_Filter_Type'Class` object. E.g., for the example filtering application, the filter is installed using:

```
use ZanyBlue.Text.Formatting;
...
App_Filter : aliased My_Filter_Type;
...
Set_Filter (App_Filter'Access);
```

There is a cost associated with filtered messages:

- The various ZanyBlue routines are called.

- Any arguments will be "boxed" and appended to an argument list.
- The filtering code is called for all messages.

This cost, however, does not included the cost associated with formatting the message as the filtering occurs before any message formatting happens.

3.7 Stub Implementations

3.7.1 Generic_Fixed

The `Generic_Fixed` package implemented formatting for the fixed floating point type. This is a stub implementation in this version of the ZanyBlue library and simply dispatches to the underlying Ada `Wide_Image` routine to format the value.

Generic ZanyBlue text packages should be instanciated at the library level to prevent run time accessibility exceptions.

3.8 ZanyBlue Formatting Implementation

The primary formatting method is the `Format` set of functions which format a message given a facility name, a key within that facility and a set of arguments (either as an `Argument_List` or as individual 'boxed" arguments). The final two arguments for both this set of `Format` functions or the `Print` and `Print_Line` procedures (explained later) is the locale and the catalog. All the functions and procedures defined in this section are defined in the package `ZanyBlue.Text.Formatting` which is generally the only ZanyBlue package needed by applications.

The locale defaults to the current locale and generally need not be specified. A possible example of where a locale would need to be specified would be a client/server application where the client sends a locale name defining their preferred locale for messages.

The use of the catalog argument is even rarer and allows messages to be defined/loaded into separate catalogs. The ZanyBlue library maintains a global common catalog which is used as the default for all functions and procedures that take catalog arguments.

Format Functions

The specification of the `Format` function is

```
function Format (Facility  : Wide_String;
                 Key       : Wide_String;
                 Arguments : Argument_List;
                 Locale    : Locale_Type := Current_Locale;
                 Catalog   : Catalog_Type := Standard_Catalog)
```

along with the in-line boxed argument version:

```
function Format (Facility  : Wide_String;
                 Key       : Wide_String;
                 Argument0 : Argument_Type'Class := Null_Argument;
                 Argument1 : Argument_Type'Class := Null_Argument;
                 Argument2 : Argument_Type'Class := Null_Argument;
                 Argument3 : Argument_Type'Class := Null_Argument;
                 Argument4 : Argument_Type'Class := Null_Argument;
                 Locale    : Locale_Type := Current_Locale;
                 Catalog   : Catalog_Type := Standard_Catalog)
   return Wide_String;
```

Print Procedures

Corresponding to the formatting functions, a set of `Print` and `Print_Line` procedures are available which print to the formatted message to the standard output file or the given file argument. These procedures have versions that take both an argument list, i.e.,

```
procedure Print (Facility  : Wide_String;
                 Key       : Wide_String;
                 Arguments : Argument_List;
                 Locale    : Locale_Type := Current_Locale;
                 Catalog   : Catalog_Type := Standard_Catalog);
```

and the in-line "boxed" arguments, e.g.,

```
procedure Print (Facility  : Wide_String;
                 Key       : Wide_String;
                 Argument0 : Argument_Type'Class := Null_Argument;
                 Argument1 : Argument_Type'Class := Null_Argument;
                 Argument2 : Argument_Type'Class := Null_Argument;
                 Argument3 : Argument_Type'Class := Null_Argument;
                 Argument4 : Argument_Type'Class := Null_Argument;
                 Locale    : Locale_Type := Current_Locale;
                 Catalog   : Catalog_Type := Standard_Catalog);
```

The signature for the `Print_Line` versions are similar. Both the `Print` and `Print_Line` procedure sets have corresponding versions that take a first argument giving the destination file, e.g.,

```
procedure Print (Destination : Ada.Wide_Text_IO.File_Type;
                 Facility    : Wide_String;
```

```
Key         : Wide_String;
Arguments   : Argument_List;
Locale      : Locale_Type := Current_Locale;
Catalog     : Catalog_Type := Standard_Catalog);
```

Plain Formatting Versions

Both the `Format` functions and `Print` procedure have versions that take a message format instead and arguments (either as an argument list or as in-line "boxed" arguments), e.g.,

```
function Format (Text      : Wide_String;
                 Arguments : Argument_List;
                 Locale    : Locale_Type := Current_Locale)
```

The locale argument is still required in this context as arguments are still formatted within the context of a locale.

Usage of these functions and procedures do not externalize the message text and, as such, do little to help internationalize applications.

Localized Exceptions

Ada allows exceptions to be raised with a message string, e.g.,

```
raise My_Exception with "Something is wrong here";
```

The ZanyBlue library includes `Raise_Exception` procedures with signatures paralleling the `Format` methods. The procedures raise the identified exception with a localized formatted messages. Since the Ada standard defines exception message to be a `String`, the formatted `Wide_String` is converted to a `String` by UTF-8 encoding the `Wide_String`. The specification of the argument list version of this procedure is

```
procedure Raise_Exception (E         : Ada.Exceptions.Exception_Id;
                      Facility     : Wide_String;
                      Key          : Wide_String;
                      Arguments    : Argument_List;
                      Locale       : Locale_Type := Current_Locale;
                      Catalog      : Catalog_Type := Standard_Catalog);
```

The conversion of `Wide_String` to an UTF-8 encoded `String` uses the GNAT specific Unicode functions.

Missing Arguments and Exceptions

Format strings refer to arguments by index, e.g.,

```
moons=There are {0} moons orbiting "{1}".
```

expects two "boxed" arguments. If supplied with less than expected, e.g.,

```
Print_Line ("myapp", "moons", +10);
```

where the planet name is not supplied, is, by default, considered an error and the exception `No_Such_Argument_Error` is raised. This behavior can be adjusted by calling the catalogs routine `Disable_Exceptions`. When exceptions are disabled, missing arguments are replaced in the formatted string with the format information enclosed in vertical bars rather than braces.

The `Disable_Exceptions` has an inverse routine `Enable_Exceptions` which re-enables exceptions. This is either on the default standard catalog or a user supplied argument catalog. The status of exceptions for a catalog can be queried using the function `Exceptions_Enabled`.

3.9 Locale Type

The `Locale_Type` defines a locale which is used to select localized messages at run time. The type basically maps to the standard ISO names used to identify the language, script and territory for the locale. Typical examples of a locale name are

1. "`fr`" for French. Here only the language abbreviation is used. Here only the language is specified.
2. "`en_US`" for English as spoken in the United Stated, where the language and territory are specified.
3. "`zh_Hant`" for Traditional Chinese, where the language and script are specified.
4. "`zh_Hans_CN`" for Simplified Chinese as spoken in China where language, script and territory are specified.

Language and territory abbreviations must be either 2 or 3 characters in length, script abbreviations must be 4 characters. For a list the various language, script and territory codes, see the source properties files in "`src/text/mesg`".

3.9.1 Locale Resolution

When accessing localized data, e.g., a message for a facility or some built-in localized data, the ZanyBlue library will perform the standard search

through the locales for the data starting with the user supplied locale (normally the standard, current, locale).

This search applies rules to move from more specific locales to more general locales walking up a virtual tree of locales until the requested data is found or the root locale is reached.

The algorithm implemented in the ZanyBlue library is a general locale parenting algorithm which does the obvious for simple language and territory locales. E.g., the parent of the locale "`de_DE`" is "`de`" which, in turn, has as it's parent the root locale. A similar algorithm is used for simple language and script locales, e.g., the parent of the locale "`en_Latn`" is "`en`", which, again, has as it's parent the root locale.

The locale parenting algorithm for full language, script and territory locales will have an ancestor tree of language and script, then language and territory, then language and finally the root locale. For example, the sequence of locales tried for the locale "`fr_Latn_FR`" is

1. `fr_Latn_FR`
2. `fr_Latn`
3. `fr_FR`
4. `fr`
5. Root Locale

This locale resolution occurs at run-time when a message for a particular facility is requested. The locale resolution for the built-in locale specific data, e.g., day names, time formats, etc., occurs at compile time. E.g., to access the name associated with Sunday requires only a few table lookups are at runtime, e.g.,

Function Call	Result
Full_Day_Name (Make_Locale ("en"), Sun)	"`Sunday`"
Full_Day_Name (Make_Locale ("fr"), Mon)	"`lundi`"
Full_Day_Name (Make_Locale ("de"), Tue)	"`Dienstag`"

Note: applications would normally just format, via message strings, values, e.g., `Time` values and let the type formatter access the lower level localized values, in this case when formatting `Time` values as dates, the localized date names might be accessed depending on the format style and locale.

It should also be noted the values returned for localized data are `Wide_Strings` and generally contain non-ASCII characters. Ad hoc testing on modern Unix

(Linux) systems using X-Windows will display the correct characters (with the cooperation of the compiler, e.g., the "`-gnatW8`" switch for GNAT). On Windows, the DOS command will will normally *not* display such character correctly. It is possible to enable display of non-ASCII characters via the selection of a code page for the command window.

3.9.2 Creating Locales

Values of `Locale_Type` are created via the `Make_Locale` function, e.g.,

```
My_Locale : constant Locale_Type := Make_Locale ("en_IE");
```

3.9.3 Changing Default Locale

The ZanyBlue Text routines allow the explicit definition of the locale for a particular function/procedure call but this normally not needed allowing the locale to default to the currently defined locale. The default locale is taken from the process environment via the variable `ZB_LANG`, and, if that is not defined, uses

1. On Unix systems, the value of the environment variable `LANG`.
2. On Windows systems, the translation of the user's default LCID value to a standard locale name (language, script and territory triple).

The default locale used can be adjusted at run time using the `Set_Locale` routine, e.g., to explicitly set the locale to Canadian French, the call would be

```
Set_Locale (Name => "fr_CA");
```

The Makefiles for the example applications generally include "`run`" targets which run the applications in the default locale. They also include rules to run application in other locales by tagging on the locale name to "`run_`", e.g., to run the `ojdbc` example in a Greek locale, the command would be

```
$ make run_el
```

3.9.4 Creating Locales

The default locale is created on application started and defaults to the locale associated with the running process. This is normally the expected locale. A locale can be explicitly created using the three variations of the `Make_Locale` function:

- By supplying a locale name, e.g., "`en`", "`en_US`" or "`en.us`".

3.9.5 Message Source Locale

The current locale is used to locate localized messages in a facility, e.g., in a French locale, messages defined by the "`_fr`" properties files would be used, if available. If not available, then the messages defined by the base properites file will be used. If the application is developed in an English environment, then the base properties file will normally contain English messages. However, it is not uncommon to have non-English developers choose to use an English base properties file.

By default, the current locale is also used to format arguments to messages. This has the biggest impact for dates and times. For example, the following message

```
Today=Today is {0,datetime,EEEE}.
```

will print the day name using the code

```
Print_Today (+Clock);
```

will, in an English locale, print the message

```
Today is Thursday.
```

If the same application is run in a French locale and a French localization is not available for the message, then the base message text is used, i.e., the English text. However, if the French locale will be used to format the date/time argument generating the message,

```
Today is jeudi.
```

Such mixed language messages should normally be avoid in applications. It is better the entire message be in English, even in non-English locales.

To support this functionality, the ZanyBlue Text library associates a locale with each message. For localized messages, this is the locale associated with the localized property file, e.g., messages in the file "`App_fr.properties`" will have the locale "`fr`" associated with them. Using the associated message locale is controlled via the enable and disable source locale routines. This is enabled by default. See the source locale text example.

For the base message file, the default locale associated with the messages is the root locale. This should normally be explicitly set using the `zbmcompile` "`-s`" option.

3.10 Codecs Type

The codecs type captures the information related to encoding and decoding internal wide Unicode strings to externally useable character sequences, i.e., narrow strings. The current implementation initializes the locale based on the encoding defined by the "`LANG`" environment varaible encoding definition. This is normally appended to the locale name separated by a period, e.g., "`en_US.UTF-8`" defines an encoding of "UTF-8".

3.11 Built-in Localizations

Localization for dates, times, language names, script names and territory names are compiled into the ZanyBlue Text library based on the Unicode.org Common Locale Date Repository (CLDR). For details on date and time formatting see the section on `Ada.Calendar.Time` argument types later.

The localized name associated with standard language, script and territory abbreviations are available via the various routines defined in the package `ZanyBlue.Text.CLDR`.

The library is currently implemented to use the CLDR data to determine the zero character when printing numbers (integers). This is normally the standard ASCII "0", however, some languages, e.g., Arabic, have there own numeric characters. In Arabic locale, `ZanyBlue.Text` will generate numeric (integer) output using Arabic numerals. Whether this is a feature or an error is unclear at this time.

For this release, the built-in locales are

Code	Language
ar	Arabic
cs	Czech
da	Danish
de	German
el	Greek
en	English
en_AU	English (Australia)
en_CA	English (Canada)
en_IE	English (Ireland)
en_GB	English (Great Britian)
en_NZ	English (New Zealand)
en_ZA	English (South Africa)
es	Spanish
fi	Finnish
fr	French
ga	Irish
he	Hebrew
hu	Hungarian
it	Italian
ja	Japanese
ko	Korean
nb	Norwegian Bokmøal
nl	Dutch
pl	Polish
pt	Portuguese
ro	Romanian
ru	Russian
sk	Slovak
sv	Swedish
th	Thai
tr	Turkish
zh	Chinese (Simplified)
zh_Hant	Chinese (Traditional)

Running the example applications, in particular the `time` example, will display localized day, month, etc. names and localized formats.

3.12 CLDR Data

The Unicode.org CLDR data used to define the locale specific information such as the date and time formats also includes localized names for languages, scripts and territories. This localized information is included in the ZanyBlue Text library via the `ZanyBlue.Text.CLDR` package and can be used to translate abbreviations, e.g, "`en`" to localized named, e.g., the function call,

```
Language_Name ("en")
```

returns "English" in an English locale and "anglais" in a French locale. There localized names for scripts and territories are available via the functions `Script_Name` and `Territory_Name` functions.

All functions take an optional `Unknown` parameter giving the result returned for unknown names (defaulting to the empty string) and a final locale parameter.

3.13 Pseudo Translations

One of the easiest mistakes to make with an internationalize application is to include hard-coded strings, i.e., not externalize the message text into a `.properties` file. One technique to detect hard-coded strings is to generate a pseudo translation in a test locale and test the application. This requires "translation" of a `.properties` file into a pseudo locale (the choice is normally Swahili in Kenya, i.e., `sw_KE`) and rebuild of a test application with the pseudo translations included.

ZanyBlue adopts a different approach and includes psuedo translation as part of the library rather than an after the fact exercise. The pseudo translation support built into the library support the translation of messages using simple wide character to wide character replacement, e.g., replace all ASCII character with their uppercase equivalents. Each message is further highlighted using start and end of message marker characters, the left and right diamond characters. Additionally, embedded arguments are surrounded by French quote characters.

To enable the built-in pseudo translations, the catalogs procedure

```
procedure Enable_Pseudo_Translations (Catalog : Catalog_Type;
                                      Mapping : Pseudo_Map_Vector);
```

can be used. The `Mapping` argument gives the character to character mapping that should be used in addition to the message and argument marking of the pseudo translation.

The mappings defined by the ZanyBlue library are:

1. `Null_Map` which preserves the message text but includes the start and end of messages and arguments.

2. `Uppercase_Map` in addition to the start and end markers for messages and arguments, convert the message text to upper case (applies only to ASCII characters).

3. `Lowercase_Map` in addition to the start and end markers for messages and arguments, convert the message text to lower case (applies only to ASCII characters).

4. `Halfwidth_Forms_Map` in addition to the start and end markers for messages and arguments, convert the message text to the halfwidth forms for Latin alphabetic and numeric characters.

5. `Enclosed_Alphanumeric_Map` in addition to the start and end markers for messages and arguments, convert the message text to the enclosed alphanumeric forms for Latin alphabetic characters.

Note: The halfwidth forms and enclosed alphanumeric mappings require the appropriate fonts be installed.

In addition to changing the characters used for the message, the Unicode character "Diamond with left half black" (U+2816) is prefixed and "Diamond with right half black" is suffixed. This allow the visual determination of where message strings begin and end. A relatively common programming error is to generate a message by concatenate a set of sub-messages. This is apparent in a psuedo translated view of the application.

Normal argument handling occurs for pseudo translated messages and the values are substituted into the message string. The text of the values are not modified by psuedo translation. Value are, however, delimited by French quotes (guillemets, chevrons). Figure **??** show the output of the `dumplocale` example application with half width psuedo translation enabled. As can be seen from the message delimiters, the header "Numeric Formats" is a multi-line messages and is displayed using the half width font. The Decimal, Scientific, etc, values are formatted arguments (as can be seen from the chevrons surrounding the values and are not displayed using the half width font.

The example applications support pseudo translation via the `-x` options.

The GPS example patches enable pseudo translation for GPS via the command line options `--pseudo=`*val*, where *val* is one of `u` (upper), `l` (lower), `h` (halfwidth) and `e` (enclosed). When halfwidth or enclosed mappings are

```
i c  F o r m a t s
- - - - - - - - - - -◆
                      »   «#,##0.###»◆
FIC                   »   «#E0»◆
                      »   «#,##0%»◆
Y                     »   «¤#,##0.00;(¤#,##0.00)»◆

i c  I t e m s
- - - - - - - - -◆
_POINT_CHARACTER »   «.»◆
```

used, the "linkage" between the standard menu item names and the localized names is lost and additional menu items are created. See figures ?? to ??.

3.14 Message 'Metrics'

Testing of internationalized applications (globalization, g11n, testing) is slightly different from normal functional testing. It is generally assumed the core functionality of the application is not dependent on the the messaging. Core testing is frequently driven by coverage data derived from the test cases. Performing testing in a globalized application to meet the same coverage to generally not something that is needed. The globalization testing needs to verify the application is fully localized in the target language and can handle localized input. This is generally a subset of the overall testing applied to an application.

To facilitate globalization testing, the library keeps track of the number of times a message is used giving a coverage number, from a message point of view. The accumulated usage information can be saved to an XML file using the `Write_Usage` routines in the text `Metrics` package.

3.15 The zbmcompile Utility

The `zbmcompile` utility compiles `.properties` files into an Ada representation allowing easier access to the message text via lookup in a catalog.

The simplest usage of the utility defines the name of the package to be created and the facility to be compiled, e.g., for the moons example application

```
$ zbmcompile -i -v Moons_Messages moons
This is ZBMCompile, Version 0.1.0 ALPHA (r1627) at 5:52 PM on 11/19/10
Copyright (c) 2009-2010, Michael Rohan.  All rights reserved
Loaded 16 messages for the facility "moons" (4 locales)
Loaded 1 facilities, 4 keys, 4 locales and 16 messages
Wrote the spec "Moons_Messages" to the file "moons_messages.ads"
Wrote the body "Moons_Messages" to the file "moons_messages.adb"
ZBMCompile completed at 5:52 PM on 11/19/10, elapsed time 0:00:00.263
```

Here the generated package `Moons_Messages` is compiled from the `.properties` files for the `moons` facility in the current directory.

3.15.1 Controlling Status Output

The `zbmcompile` utility supports options to control the amount of status information printed:

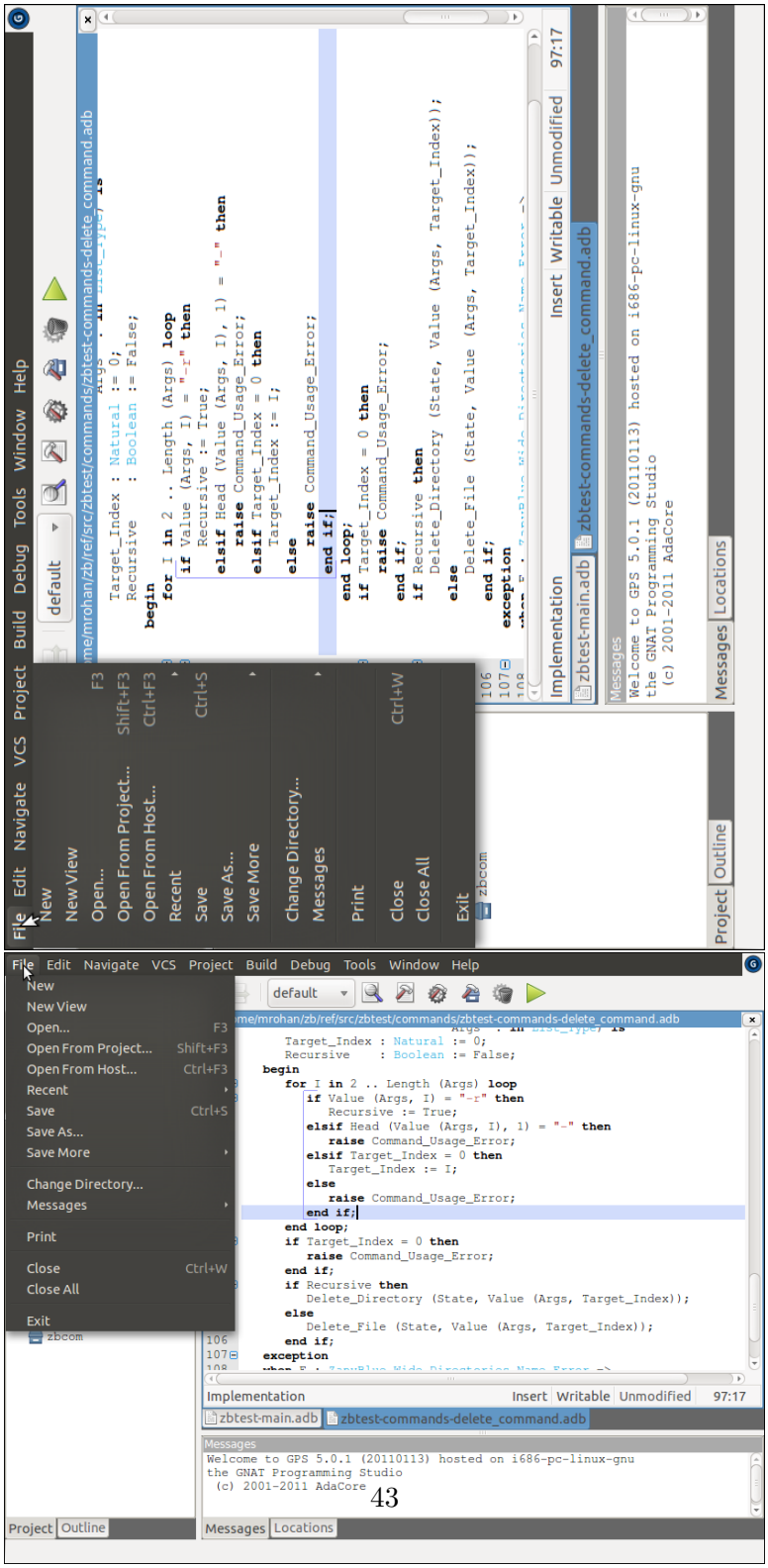


Figure 3.3: GPS display with no pseudo translation.

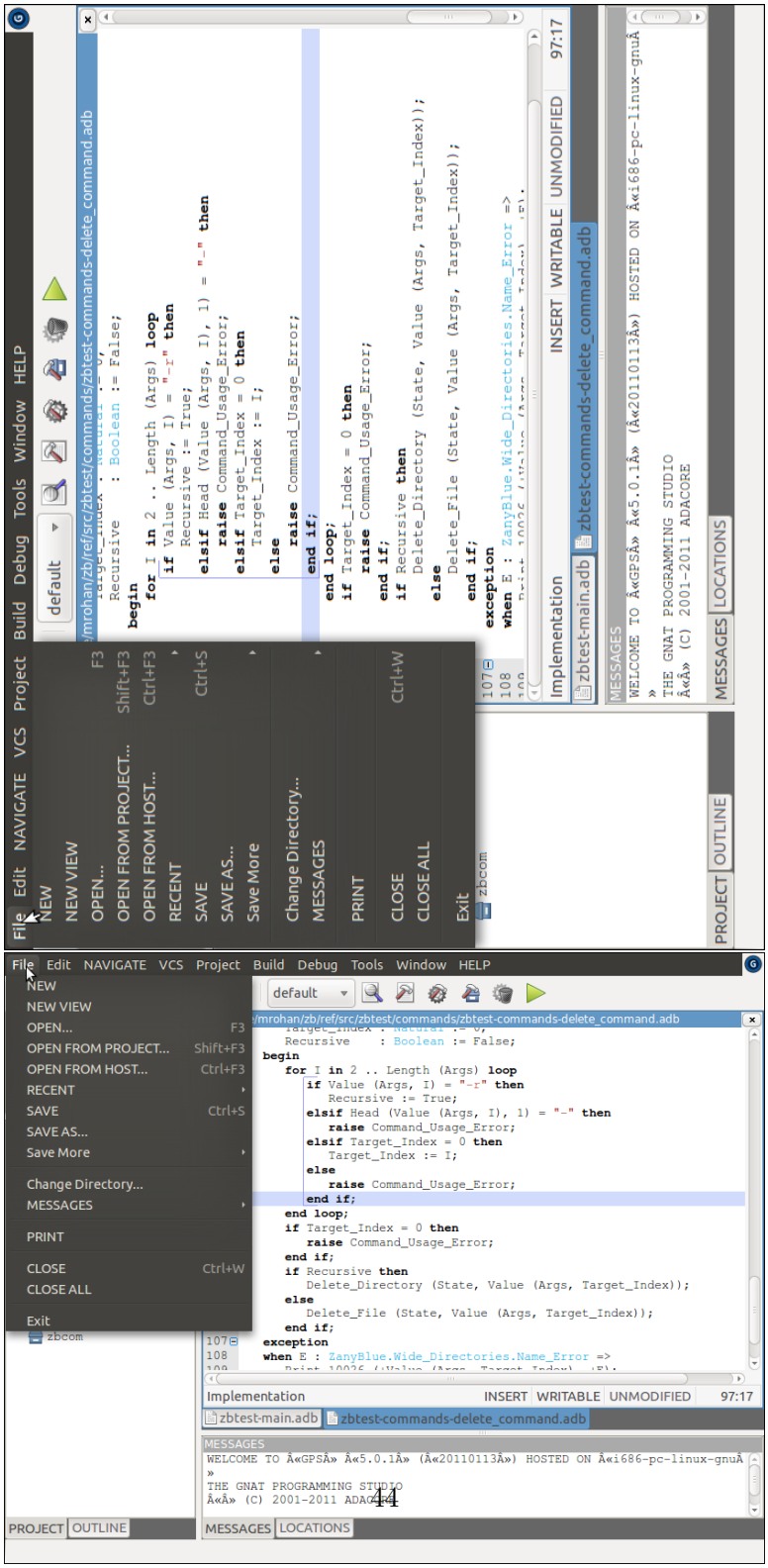
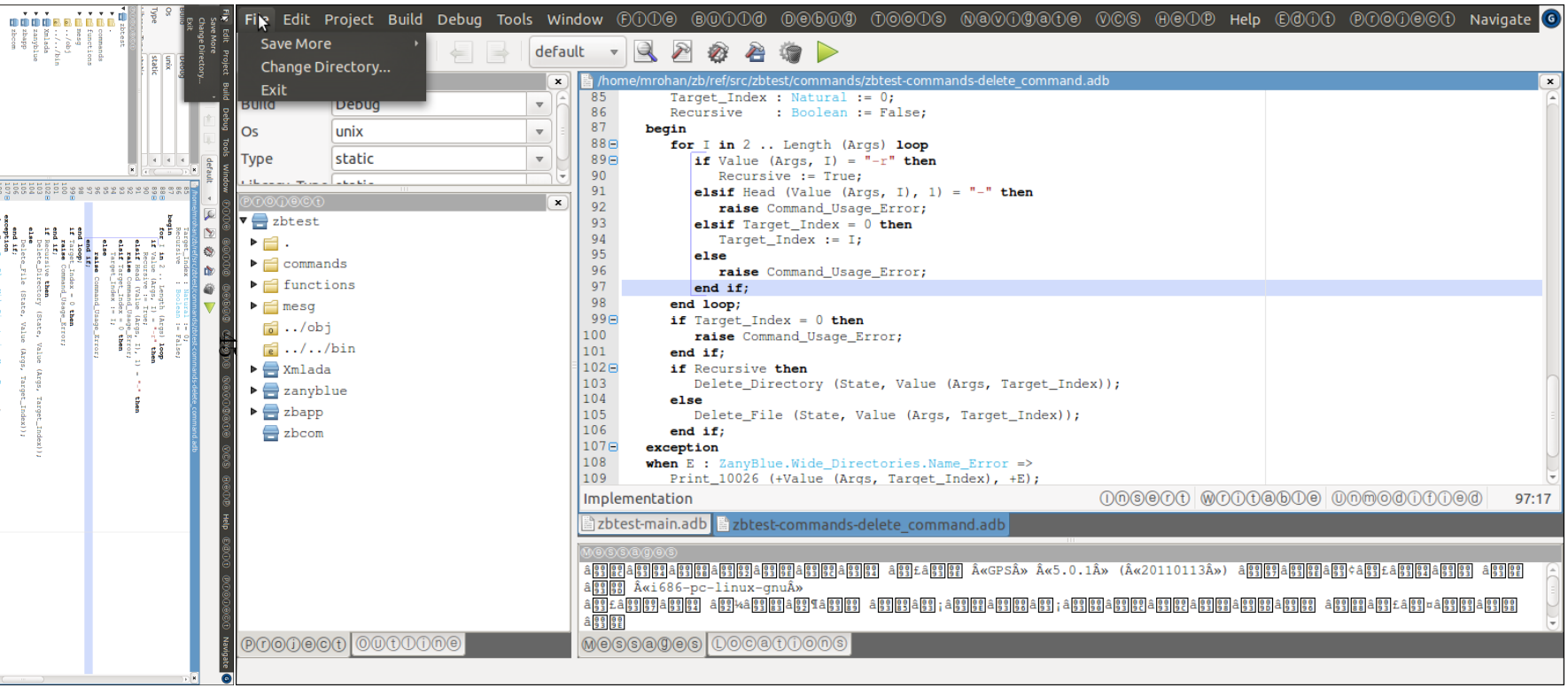


Figure 3.4: GPS display with uppercase pseudo translation.


File Edit Project Build Debug Tools Window Help
Save More
Change Directory...
Exit
default
Build
Debug
Os
unix
Type
static
Project
zbtest
.
commands
functions
mesg
../obj
../../bin
Xmlada
zanyblue
zbapp
zbcom
Project Outline
/home/mrohan/zb/ref/src/zbtest/commands/zbtest-commands-delete_command.adb
85 Target_Index : Natural := 0;
86 Recursive : Boolean := False;
87 begin
88 for I in 2 .. Length (Args) loop
89 if Value (Args, I) = "-r" then
90 Recursive := True;
91 elsif Head (Value (Args, I), 1) = "-" then
92 raise Command_Usage_Error;
93 elsif Target_Index = 0 then
94 Target_Index := I;
95 else
96 raise Command_Usage_Error;
97 end if;
98 end loop;
99 if Target_Index = 0 then
100 raise Command_Usage_Error;
101 end if;
102 if Recursive then
103 Delete_Directory (State, Value (Args, Target_Index));
104 else
105 Delete_File (State, Value (Args, Target_Index));
106 end if;
107 exception
108 when E : ZanyBlue.Wide_Directories.Name_Error =>
109 Print_10026 (+Value (Args, Target_Index), +E);
Implementation
97:17
zbtest-main.adb
zbtest-commands-delete_command.adb
Messages
Locations

`-q` Reduced the amount of output to just error and warning messages.

`-v` Increase the amount of output generated.

`-D` Increase the amount of output to aid debugging.

3.15.2 Definition of Properties Directory

The default usage assumes the `.properties` files are located in the current directory. To locate the files in another directory, the `-d` option can be used, e.g.,

```
$ zbmcompile -i -v -d mesg Moons_Messages moons
```

would locate the properties associated with the `moons` facility in the `mesg` directory.

3.15.3 Properties File Extension

The default file extension used when locating properties files is `.properties`. This can be change using the `-e` option. E.g., to load all `.msg` files,

```
$ zbmcompile -i -v -e .msg Moons_Messages moons
```

3.15.4 Auto-Initialization

The generate Ada code include an initialization routine which loads the messages into a catalog (defaulting to the standard global catalog). The `-i` option includes a call to this initialization procedure in the body of the generated package. This allows the inclusion of the message in an application by simply including the specification in a compilation unit, normally the main unit. The option also causes the inclusion of a warning suppression pragma in the specification to allow compilation in a strict compilation environment.

3.15.5 Optimization of Messages

When the `zbmcompile` loads facilities in sequence which, in general, distributes the messages associated with various locales. The `zbmcompile` optimize mode, the `-O` option, performs a second pass on the loaded messages gathering messages for each locale together. Since applications generally don't change locale very often, if at all, having all the message strings for a locale located in the same set of pages can improve performance.

For symmetry reasons, the `-g` option is included which disables optimization.

3.15.6 Locale Selection

Occasionally, only a subset of the `.properties` files should be compiled into the generated Ada package. This selection is supported using the `-B`, for base locale, and `-L` options. E.g., to generate the Moons message package for just the base language, French and German, the command would be

```
$ zbmcompile -v -B -L fr -L de Moons_Messages moons
```

These option could be used to generate language packs, possibly via shared library or dll implementations.

3.15.7 Forced Compilation

When testing using messages from Java projects, the message files will frequently be found to contain errors from a `zbmcompile` point of view. To force the generation of Ada in the context of input errors, the `-F` can be used. Note, when used, there is no guarantee the resultant generated Ada code will compile.

3.15.8 Definition of External Initialization Routine

The possible direction for Ada localization is to allow the loading of language pack at run-time via shared library or dlls. This has not been investigated but in the context of dynamic loading of shared libraries or dll's, having an initialization name that is well defined makes the implementation easier. To support this, the untested functionality of supplying a linker name for the initialization routine is allowed via the `-x` option, e.g.,

```
$ zbmcompile -v -B -x "moons_messages" Moons_Messages moons
```

3.15.9 Generation of Accessor Packages

The generation of message accessor packages creating routines for each message defined with parameter lists matching the arguments defined by the message text is controlled via the two options `-a` and `-G`.

The first style, `-a`, generates all accessor style packages:

1. `exceptions`, generate routines to raise exceptions with localized message strings (wide strings converted to encoding associated with the locale).

2. `strings`, generate functions returning locale encoded strings for the localized messages.

3. `wstrings`, generate functions returning wide strings for the localized messages.

4. `prints`, generate routines printing the localized messages to files as locale encoded strings.

5. `wprints`, generate routines printing the localized messages to files as wide strings (wide files).

The `-G` option allows the selection of individual accessor style packages, e.g., for an application that only uses messages to raise exceptions and print to wide files, the command line options would be `-G exceptions` and `-G wprints`, i.e., the `-G` option can used multiple time on the same command line.

The packages generated are child packages of the primary package given on the command line with names based on the facility name, e.g., if the facility name is "Moons", the generated child packages would be

Style	Child Package Name
`exceptions`	`Moons_Exceptions`
`strings`	`Moons_Strings`
`wstrings`	`Moons_Wide_Strings`
`prints`	`Moons_Prints`
`wprints`	`Moons_Wide_Prints`

3.15.10 Output Directory

By default, the generated packages are written to the current directory. To select a different directory, the `-o` option can be used, e.g.,

```
$ zbmcompile -o mesg Moons_Messages moons
```

would write the packages to the `mesg` directory. The directory must already exist.

3.15.11 Adjusting the Generated Code

The `zbmcompile` command has a number of options used to adjust the generated code.

Comments for Accessors

By default, the generated routines for accessors include the base message text as a comment. This allows the display of the text of the messages within GPS and makes browsing the source easier. These comments can be suppressed using the `-C` option. One reason to suppress these comments would be to minimize recompilations when updating messages. The `zbmcompile` command will only create new source files if the generated contents differs from the existing files (or the files currently doesn't exist). With comments suppressed, updating message strings would only result in the update to the primary package body requiring, in general, a single recompilation and relink of an application. With comment enabled, the accessor spec files would be updated resulting in larger recompilations.

Argument Modes

The default code generated does not include the `in` keyword for in routine arguments. Some code bases might have style rules requiring explicit use of this keyword. To require the generated code include this keyword, the `-m` option can be used.

Positional Elements

The generated code includes a number of initialized tables (arrays). The default style for such tables is simply to list the entries using implicit index association. Again, some code bases might require that explicit numbering of such code. This can be enabled using the `-p` command line option. E.g., instead of

```
Facilities : constant ZT.Constant_String_List (1 .. 7) := (
                Facility_1'Access,
                Facility_2'Access,
...
```

the generated code would be

```
Facilities : constant ZT.Constant_String_List (1 .. 7) := (
                1 => Facility_1'Access,
                2 => Facility_2'Access,
...
```

Output Line Lengths

The generated code keeps within the standard 80 column style for source files. There are two control parameters which can be used as arguments to the `-T` command line option to adjust this for selected items:

1. The accumulated message strings are stored as a single constant string initialized using a multi-line string. The length of the substrings written per line can be controlled using the command line `pool` item with an integer value, e.g., `-T pool 30` to reduce the size.

2. The base message text written as a comment on accessors is wrapped to ensure the 80 column limit is not exceeded. This results in wrapping within words. To increase the limit for messages, use the `comment` item, e.g., `-T comment 120`. Accessor comments include breaks for messages with new lines.

3.15.12 Consistency Checks

Localized messages are cross checked with the base locale messages to ensure they are consistent, i.e., it is an error for a localized message to refer to an argument not present in the base message.

There are two options that control the consistency checks performed

1. `-u`, disable the consistency checks. This is a rather dangerous thing to do and should be avoided.

2. `-r` define the base locale. Normally the base locale messages are in the properties file without a locale, i.e., the root locale. Some applications might choose to use explicit locale naming for all properties files. The `-r` option can be used to designate which of the available locales is the base locale.

3.15.13 Selection of Source Locale

The source locale for the base properties file can be specified using the `-s` option. See **??** for details on this functionality.

3.15.14 Stamp File

For Makefile base builds, dependency checking is simplified if a single fixed named file can be used rather than the set of generated spec and body files normally created by `zbmcompile`. The `-S` option allows the definition of a simple time stamp file which is updated whenever the `zbmcompile` command is run. This file can be used to define built dependencies in Makefiles.