Advanced Resource Embedder

Stephane Carrez

2024-02-10

Contents

1 Introduction

Incorporating files in a binary program can sometimes be a challenge. The `Advance Resource Embedder` is a flexible tool that collects files such as documentation, images, scripts, configuration files and generates a source code that contains these files. It is able to apply some transformations on the collected files:

- it can run a Javascript minifier such as `closure`,
- it can compress CSS files by running `yui-compressor`,
- it can compress files by running `gzip` or another compression tool.

Once these transformations are executed, it invokes a target generator to produce a source file either in C, Ada or Go language. The generated source file can then be used in the final program and taken into account during the compilation process of that program. At the end, the binary will contain the embedded files with their optional transformations.

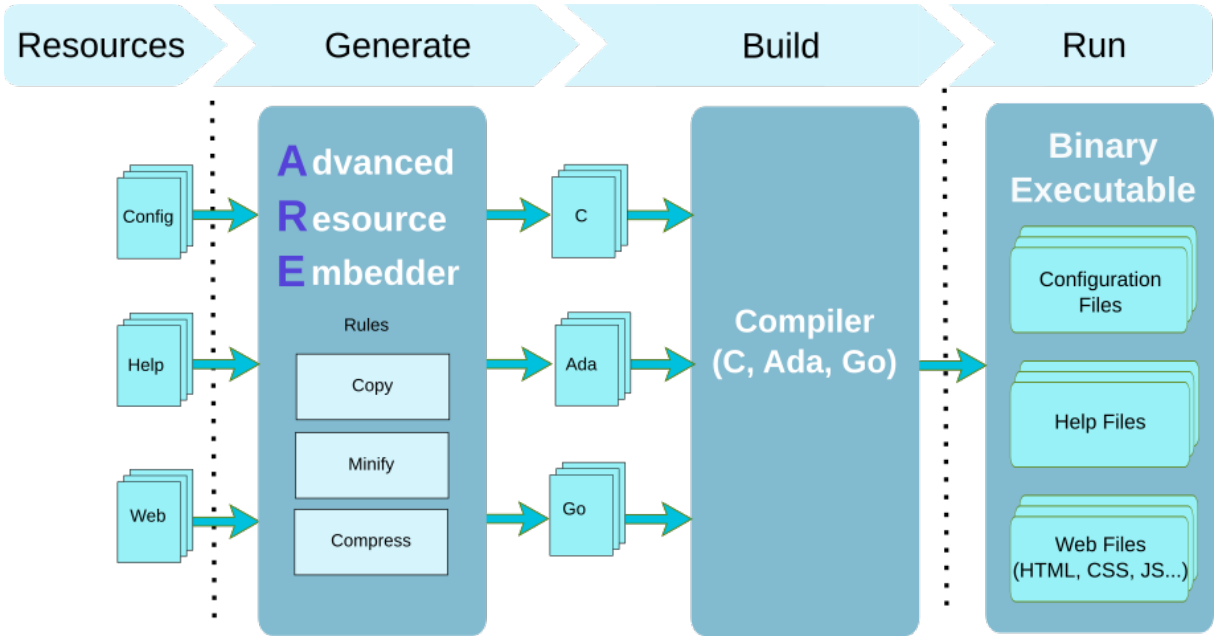

Figure 1: Resource Embedder Overview

The process to use ARE is simple:

- You describe the resources that you want to embed. The description is either made on command line arguments or by writing an XML file. The XML description gives more flexibility as it allows to define a transformation rule that must be executed on the original file before being embedded.

This allows to minify a Javascript or CSS file, compress some files and even encrypt a file before its integration.

- You run the ARE command with your target language and rule description and you give the tool a list of directories that must be scanned to identify the files that must be collected. The ARE tool scan the directories according to the patterns that you have given either on the command line or in the XML rule description. After identifying the files, the tool applies the rules and execute the transformations. The ARE tool then invokes the target language generator that writes one or several files depending on the list of resources.
- Once the files are generated, you use them in your program and add them in your build process as they are now part of your sources. After building your program, it now embeds the resource files that were collected and optionally transformed.

This document describes how to build the tool and how you can use the different features to embed files in a binary program witten in Ada, C/C++ or Go.

2 Installation

This chapter explains how to build and install the tool.

2.1 Using Alire

The `Advanced Resource Embedder` is available as an Alire crates to simplify the installation and setup your project. Run the following commands to setup your project to use the library:

```
alr index --update-all
alr with are
```

2.2 Without Alire

2.2.1 Before Building

To build `Advanced Resource Embedder` you will need the GNAT Ada compiler, either the FSF version available in Debian, FreeBSD systems NetBSD or the AdaCore GNAT Community 2021 edition. Because there exists different versions of the compiler, you may have to adapt some of the commands proposed below for the installation.

2.2.1.1 Ubuntu 22.04

Install the following packages:

```
sudo apt install -y make git
sudo apt install -y gnat-10 gprbuild libxmlada-dom10-dev
```

2.2.1.2 FreeBSD 13

Install the following packages:

```
pkg install gmake gnat12 gprbuild git
```

2.2.1.3 Windows

Get the Ada compiler from AdaCore Download site and install.

Install the following packages:

```
pacman -S git
pacman -S make
pacman -S base-devel --needed
```

2.2.2 Getting the sources

The project uses a sub-module to help you in the integration and build process. You should checkout the project with the following commands:

```
git clone --recursive https://gitlab.com/stcarrez/resource-embedder.git
cd resource-embedder
```

2.2.3 Configuration (optional)

Running the `configure` script is optional and is useful if the pre-defined default configuration must be changed.

The `configure` script is used to detect the build environment, setup specific build configuration. 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:

- `--enable-distrib` to build for a distribution and strip symbols,
- `--disable-distrib` to build with debugging support,
- `--enable-coverage` to build with code coverage support (`-fprofile-arcs -ftest-coverage`),
- `--prefix=DIR` to control the installation directory,
- `--help` to get a detailed list of supported options.

In most cases you will configure with the following command:

```
./configure
```

2.2.4 Build

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

```
make
```

If you have an old GNAT compiler (gcc < 8) you may build with:

```
make GNAT_SWITCH=NO_CALLBACK
```

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

```
make test
```

2.2.5 Installation

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

```
make install
```

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

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

3 Using Advanced Resource Embedder

To embed files and generate Ada, C or Go source file, the `Advanced Resource Embedder` must identify the files, organize them and may be perform some transformation on these files before their integration. To control this process, it is possible to use some options passed to the `are (1)` tool but a better control is achieved by using an XML configuration file.

3.1 Defining resources

The XML file describes a list of resources that must be generated. It is introduced by the **package** root XML element and each resource is represented by a `resource` XML element. A resource is assigned a name and composed of several installation rules that describe how files are integrated and whether some transformations are made before their integration.

```
<package>
  <resource name='Help' format='string'>
    <install mode='xxx'>
      ...
    </install>
  <resource>
  <resource name='Config' format='lines'>
    <install mode='xxx'>
      ...
    </install>
  </resource>
  <resource name='Web' format='binary'>
    <install mode='xxx'>
      ...
    </install>
    <install mode='yyy'>
      ...
    </install>
  </resource>
  ...
</package>
```

The resource content can be available in several formats by the code generator. This format is controlled by the `format` attribute. The following data formats are supported:

- `binary` format provides the file content as a binary data.
- `string` format provides the file content as string.
- `lines` format splits the content in several lines and according to a set of customisable rules.

To help you in the control of the generated code, the resource description can also define specific attributes that allow you to tune the code generator. The following XML definition:

```
<package>
  <resource name='Help'
            type="man_content"
            function-name="man_get_help_content">
    <install mode='copy'>
      <fileset dir="help">
        <include name="**/*.txt"/>
      </fileset>
    </install>
  </resource>
  ...
</package>
```

creates a resource named `Help` and composed of text files located in the `help` directory and with the `.txt` file extension. The code generator will use the name `man_content` for the data type that represents the file description and it will use `man_get_help_content` for the generated function name.

3.2 Controlling the lines format

The `lines` format tells the code generator to represent the content as an array of separate lines. For this integration, some control is available to indicate how the content must be split and optionaly apply some filter on the input content. These controls are made within the XML description by using the `line-separator` and `line-filter` description: The `line-separator` indicates the characters that represent a line separation. There can be several `line-separator` definition. The `line-filter` defines a regular expression that when matched must be replaced by an empty string or a specified content. The `line-filter` are applied in the order of the XML definition.

The example below is intended to integrate an SQL scripts with:

- a separate line for each SQL statement,
- remove spurious empty lines and SQL comments.

The SQL statements are separated by `;` (semi-colon) and the `line-separator` indicates to split lines on that character. By splitting on the `;`, we allow to have an SQL statement on multiple lines.

```
<package>
  <resource name='Scripts'
            format='lines' keep-empty-lines="no"
            type='access constant String'>
    <line-separator>;</line-separator>

    <!-- Remove new lines -->
    <line-filter>[\r\n]</line-filter>

```

```
10      <!-- Remove C comments -->
11      <line-filter>/\*[^/]*\*/</line-filter>
12
13      <!-- Remove contiguous spaces after C comments removal -->
14      <line-filter replace=' '>[ \t][ \t]+</line-filter>
15
16      <install mode='copy' strip-extension='yes'>
17        <fileset dir="sql">
18          <include name="**/*.sql"/>
19        </fileset>
20      </install>
21    </resource>
22  </package>
```

Then the first `line-filter` will remove the `\r` and `\n` characters.

The regular expression `/\*[^/]*\*/` matches a C style comment and remove it.

The last `line-filter` replaces multiple tabs and spaces by a single occurence.

By default an empty line is discarded. This behavior can be changed by using the `keep-empty-lines` attribute and setting the value to **`true`**.

3.3 Selecting files

An important step in the configuration of the `Advanced Resource Embedder` is the selection of files that will be embedded. The mechanism to select files is heavily inspired by the `ant (1)` Java builder with the notion of filesets and patterns.

A fileset describes a collection of files stored in a directory and it uses a set of inclusion and exclusion patterns to select files of that directory. A fileset is described by the `fileset` XML element and it can contain several `include` and `exclude` XML element. Each `include` element describes a pattern that the file must match to be taken into account. Sometimes a file can be matched but you want to exclude it and you will use the `exclude` XML element to reject that file.

A pattern a either a fixed relative path or it may contain wildcards. A single wildcard pattern applies only to a single directory and the special notation `**/` indicates to match any child directory.

The following definition:

```
1  <fileset>
2    <include name="*.html"/>
3    <include name="*.css"/>
4    <include name="*.js"/>
5    <exclude name="test.js"/>
6  </fileset>
```

will select files from the directories passed to the `are` tool and it takes into account only files with `.html`, `.css` and `.js` extension. Child directories are excluded as well as the `test.js` file if it exists.

A fileset can indicate a directory name by using the `dir` attribute. In that case, the file selection will start from the directory with the given name.

```
<fileset dir='web'>
  <include name="**/*.html"/>
  <include name="**/*.css"/>
  <include name="**/*.js"/>
  <exclude name="preview/**"/>
</fileset>
```

That definition scans the `web` directory for each argument passed to the `are` tool and selects recursively all `.html`, `.css` and `.js` files. If the `web` directory contains a `preview` directory, that directory and any file it contains will be excluded.

You may include and combine several `fileset` XML element to describe complex file selection.

3.4 Integration modes

The `Advanced Resource Embedder` provides several modes for the integration of a file. After files are matched, a decision must be made on the files to integrate them in the output. Sometimes it happens that several source files will correspond to a single output. For this integration, it is possible to make some specific transformations.

The installation rule is described by the `install` XML element. That rule in fact contains the fileset that indicates the files that must be taken into account by the installation rule.

```
<install mode='copy'>
  <fileset>
    <include name='**/*.txt'/>
  </fileset>
</install>
```

The installation modes are described more into details in the Rules chapter.

3.5 Custom headers

It is possible to add custom headers in the generated files by using the `header` XML element within each `resource`. Each `header` element is written verbatim in the output code. It can contain comment or some target source code. A `type` attribute can be defined to limit in which file the header content

is written. By default, the header line is written in the specification (`.ads`, `.h`) and body files (`.adb`, `.c`).

```
<package>
  <resource ...>
    <header>...</header>
    <header type='spec'>...</header>
    <header type='body'>...</header>
    ...
  </resource>
  ...
</package>
```

3.6 Man page

3.6.1 NAME

are - Resource embedder to include files in Ada, C/C++, Go binaries

3.6.2 SYNOPSIS

are [-v] [-vv] [-V] [–tmp *directory*] [-k] [–keep] [-o *directory*] [-l *lang*] [–rule *path*] [–resource *name*] [–fileset *pattern*] [–ignore-case] [–list-access] [–var-access] [–var-prefix *prefix*] [–no-type-declaration] [–type-name *name*] [–function-name *name*] [–member-content *name*] [–member-length *name*] [–member-modtime *name*] [–member-format *name*] [–preelaborate] [–content-only] directory…

3.6.3 DESCRIPTION

are is a tool to generate C, Ada or Go source allowing to embed files in a binary program by compiling and linking with the compiled generated sources.

The process to use **are** is simple and composed of three steps:

- First, you describe the resources that you want to embed. The description is either made on command line arguments or by writing an XML file. The XML description gives more flexibility as it allows to define a transformation rule that must be executed on the original file before being embedded. This allows to minify a Javascript or CSS file, compress some files and even encrypt a file before its integration.
- You run the **are** command with the your target language and rule description and you give the tool a list of directories that must be scanned to identify the files that must be collected. The **are**

tool scan the directories according to the patterns that you have given either on the command line or in the XML rule description. After identifying the files, the tool applies the rules and execute the transformations. The **are** tool then invokes the target language generator that writes one or several files depending on the list of resources.

- Once the files are generated, you use them in your program and add them in your build process as they are now part of your sources. After building your program, it now embeds the resource files that were collected and optionally transformed.

The identification of files is made by using fileset patterns similar to the **ant**(1) tool. The patterns are applied to the directories that are passed to the **are** tool. Files that match the pattern are selected and taken into account. The pattern can be an exact relative path definition or it may contain wildcards. Below are some examples:

.txt This pattern matches all files with a .txt* extension in the directories passed to the command. Only the root directories are taken into account (the *.txt* files in sub-directories are ignored).

/.txt The /* pattern indicates that the pattern is applied on directories recursively. The files must then match the **.txt* pattern to be taken into account. Therefore, the** /.txt* pattern will match all *.txt* files in any directory.

config/*.conf This pattern will match the* .conf* files in the *config* directory.

web/index.html This pattern matches a fixed path.

3.6.4 OPTIONS

The following options are recognized by **are**:

-V Prints the *are* version.

-v Enable the verbose mode.

-vv Enable debugging output.

–tmp **directory** Use the directory to build the resource files. The default directory is *are-generator* and it is created in the current working directory. This option allows to choose another path.

–keep Keep the directory used to prepare the resource files. By default the *are-generator* directory (which can be overriden by the *–tmp* option) is removed when the code generation is finished. By keeping the directory, you can have a look at the files and their transformations.

–output **directory** Set the output directory path where generators writes the code.

–lang **language** Select the target generator language. The supported languages are **Ada**, **C**, and **Go**. The default language is **Ada**.

–rule **path** Read the XML file that describes the resources to generate. The use of a XML resource file allows to use the advance features of the tool such as doing some transformations on the input files. The XML resource file can describe several resources and provides mechanisms to control the generation for each of them.

–resource **name** Define the name of the resource collection. This option is used to create a resource with the given name.

–fileset **pattern** Define the pattern to match files for the resource collection. After the *–resource* option, this indicates the pattern to match the files for that resource.

–name-access Generate support to query content with a name. The code generator will declare and genrate a function which given a name returns the embedded content if that name is known.

–list-access Generate support to list the content names. Most code generator will declare a variable that represents a sorted list of names which represents the resource. It is possible to use a dichotomic search on that name array.

–var-access Declare a variable to give access to each content. When this option is given, the code generator will emit a global variable declaration with the name of the file. By using the global variable, the program can access the resource directly.

–var-prefix **prefix** Defines the prefix to be used for the variable declarations that give access access to each content. This option implies the *–var-access* option.

–no-type-declaration Do not declare any type in the package specification. It is assumed that the types used by the generated code is declared somewhere else and is visible during the compilation.

–type-name **name** Define the name of the type used to hold the information. This is the name of the C, Ada or Go type that is generated. It must be a valid name of the target language.

–member-content **name** Define the name data structure member holding the content.

–member-length **name** Define the name data structure member holding the length.

–member-modtime **name** Define the name data structure member holding the modification time.

–member-format **name** Define the name data structure member holding the content format

–preelaborate This option is recognized by the Ada generator and it tells it to emit a pragma Preelaborate in the generated specification file.

–content-only This option is specific to the Ada generator and instructs the generator to only give access to the content.

3.6.5 RULE DESCRIPTION

The rule descritions are best expressed by using an XML file. The XML file can describe several resources and for each of them it defines the files that must be included with their optional transformation. The XML file must have a *package* root element.

A resource is described by the *resource* XML element with a mandatory *name* attribute that indicates the name of the resource. It then contains an *install* XML element which describes the installation rule with the patterns that identify the files.

```
<package>
 <resource name='help' format='string'>
  <header>--  Some header comment</header>
  <install mode='copy'>
    <fileset dir='help'>
     <include name='**/*.txt'/>
    </fileset>
  </install>
 </resource>
</package>
```

A resource can be represented as an array of strings by using the *lines* format. In that case, a *line-separator* XML element indicates the character on which lines are split. The *keep-empty-lines* attribute controls whether an empty line is kept or must be discarded. The default will discard empty lines. With the *lines* format, the final content will be represented as an array of strings.

```
<package>
 <resource name='help' format='lines' keep-empty-lines='true'>
  <line-separator>\\r</line-separator>
  <line-separator>\\n</line-separator>
  <install mode='copy'>
    <fileset dir='help'>
     <include name='**/*.txt'/>
    </fileset>
  </install>
 </resource>
</package>
```

The special format *map* reads the content of files which are collected and produce a mapping table with them. The files can be a JSON file with name/value pairs and the mapping table will provide an efficient conversion of a name into the corresponding value.

```
<package>
 <resource name='Extensions_Map' format='map'
           type='access constant String'>
  <mapper type='json'/>
  <install mode='copy'>
    <fileset dir='.'>
```

```
      <include name='**/*.json'/>
    </fileset>
  </install>
 </resource>
</package>
```

3.6.6 INSTALL MODES

The **are** tool provides several installation modes:

copy Copy the file.

copy-first Copy the first file.

exec Execute a command with the file.

copy-exec The file is copied and a command is then executed with the target path for some transformations.

concat The files that match the pattern are concatenated.

bundle This mode concern Java like property files and allows to do some specific merge in the files.

merge This mode concern Java like property files and allows to do some specific merge in the files.

3.6.7 SEE ALSO

ant(1), **gprbuild(1)**, **gzip(1)**, **closure(1)**, **yui-compressor(1)**

3.6.8 AUTHOR

Written by Stephane Carrez.

4 Rules

The `Advanced Resource Embedder` provides several mechanisms to integrate files in the generated code.

An XML file file contains a set of rules which describe how to select the files to include in the generator. The XML file is read and resource rules introduced by the `resource` XML element are collected.

The source paths are then scanned and a complete tree of source files is created. Because several source paths are given, we have several source trees with possibly duplicate files and names in them.

The source paths are matched against the resource rules and each installation rule is filled with the source files that they match.

The resource installation rules are executed in the order defined in the **`package`**`.xml` file. Each resource rule can have its own way to make the installation for the set of files that matched the rule definition. A resource rule can copy the file, another can concatenate the source files, another can do some transformation on the source files and prepare it before being embedded and used by the generator.

4.1 Install mode: copy and copy-first

The `copy` and `copy-first` mode are the simpler distribution rules that only copy the source file to the destination. The rule is created by using the following XML definition:

```
1 <install mode='copy'>
2   <include name="*.txt"/>
3 </install>
```

If the tool is called with several directories that contain a same file name then the `copy` installer will complain because it has two source files for a same destination name. When this happens, you may instead use the `copy-first` mode which will take into account only the first file found in the first directory.

By default the relative path name of the file is used to identify the embedded content. Sometimes, you may want to drop the file extension and access the content by using only the name of the file without its extension. This is possible by setting the `strip-extension` attribute to `yes` as follows:

```
1 <install mode='copy' strip-extension='yes'>
2   <install name="*.txt"/>
3 </install>
```

If the file has the name `help.txt`, then it is known internally by the name `help`.

4.2 Install mode: concat

The `concat` mode provides a distribution rule that concatenates a list of files. The rule is created by using the following XML definition:

```
<install mode='concat' source-timestamp='yes'>
  <include name="NOTICE.txt"/>
</install>
```

This rule is useful when the tool is invoked with several directories that contain files with identical names. Unlike the `copy` and `copy-first` rules that take into account only one source file, the `concat` mode handles this situation by concatenatating the source files.

By default the generated file has a timestamp which correspond to the time when the `are` command is executed. By setting the `source-timestamp` attribute to **`true`**, the generated file is assigned the timestamp of the newest file in the source files.

4.3 Install mode: exec and copy-exec

The `exec` and `copy-exec` mode are the most powerful installation rules since they allow to execute a command on the source file. The `copy-exec` will first copy the source file to the destination area and it will execute the command. The rule is created by using the following XML definition:

```
<install mode='exec' dir='target' source-timestamp='true'>
  <command slow='false' output='...'>cmd #{src} #{dst}</command>
  <fileset dir="source">
    <include name="**/*"/>
  </fileset>
</install>
```

The command is a string which can contain EL expressions that are evaluated before executing the command. The command is executed for each source file. The following EL variables are defined:

Name	Description
src	defines the absolute source path
dst	defines the target destination path
name	defines the relative source name (ie, the name of the resource file)

4.4 Install mode: bundles

The `Are.Installer.Bundles` package provides distribution rules to merge a list of bundles to the distribution area. The rule is created by using the following XML definition:

```
<install mode='bundles' source-timestamp='true'>
  <fileset dir='bundles'>
    <include name="**/*.properties"/>
  </fileset>
</install>
```

4.5 Install mode: webmerge

The `webmerge` distribution rule is intended to merge Javascript or CSS files which are used by XHTML presentation files. It requires some help from the developer to describe what files must be merged. The XHTML file must contain well defined XML comments which are used to identify the merging areas. The CSS file merge start section begins with:

```
<!-- ARE-MERGE-START link=#{contextPath}/css/target-merge-1.css -->
```

and the Javascript merge start begings with:

```
<!-- ARE-MERGE-START script=#{contextPath}/js/target-merge-1.js -->
```

The merge section is terminated by the following XML comment:

```
<!-- ARE-MERGE-END -->
```

Everything withing these XML comments is then replaced either by a `link` HTML tag or by a `script` HTML tag and a file described either by the `link`= or `script`= markers is generated to include every `link` or `script` that was defined within the XML comment markers. For example, with the following XHTML extract:

```
<!-- ARE-MERGE-START link=#{contextPath}/css/merged.css -->
<link media="screen" type="text/css" rel="stylesheet"
      href="#{contextPath}/css/awa.css"/>
<link media="screen" type="text/css" rel="stylesheet"
      href="#{jquery.uiCssPath}"/>
<link media="screen" type="text/css" rel="stylesheet"
      href="#{jquery.chosenCssPath}"/>
<!-- ARE-MERGE-END -->
```

The generated file `css/merged.css` will include `awa.css`, `jquery-ui-1.12.1.css`, `chosen.css` and the XHTML will be replaced to include `css/merge.css` only by using the following XHTML:

```
<link media='screen' type='text/css' rel='stylesheet'
      href='#{contextPath}/css/merged.css'/>
```

To use the `webmerge`, the **`package.xml`** description file should contain the following command:

```
<install mode='webmerge' dir='web' source-timestamp='true'>
   <property name="contextPath"></property>
   <property name="jquery.path">/js/jquery-3.4.1.js</property>
   <property name="jquery.uiCssPath">/css/redmond/jquery-ui-1.12.1.css
      </property>
   <property name="jquery.chosenCssPath">/css/jquery-chosen-1.8.7/
      chosen.css</property>
   <property name="jquery.uiPath">/js/jquery-ui-1.12.1</property>
   <fileset dir="web">
      <include name="WEB-INF/layouts/*.xhtml"/>
   </fileset>
</install>
```

The merging areas are identified by the default tags `ARE-MERGE-START` and `ARE-MERGE-END`. These tags can be changed by specifying the expected value in the `merge-start` and `merge-end` attributes in the `install` XML element. For example, with

```
<install mode='webmerge' dir='web' source-timestamp='true'
         merge-start='RESOURCE-MERGE-START'
         merge-end='RESOURCE-MERGE-END'>
</install>
```

the markers becomes:

```
<!-- RESOURCE-MERGE-START link=#{contextPath}/css/target-merge-1.css
   -->
<!-- RESOURCE-MERGE-END -->
```

5 Generator

The code generators are invoked when the installer has scanned the directories, selected the files and applied the installation rules to produce the content that must be embedded.

5.1 Ada Generator

The Ada code generator produces for each resource description an Ada package with the name of that resource. Sometimes, the Ada package specification is enough and it contains all necessary definitions including the content of files. In other cases, an Ada package body is also generated and it contains the generated files with a function that allows to query and retrieve the file content. The Ada code generator is driven by the resource description and also by the tool options.

The code generator supports several data format to access the file content.

Format	Ada type
binary	access constant Ada.Streams.Stream_Element_Array
string	access constant String
lines	array of access constant String
map	access constant String

When the `--content-only` option is used, the code generator uses the following type to describe a file content in the `binary` format:

```
type Content_Access is
   access constant Ada.Streams.Stream_Element_Array;
```

for the `string` format it defines:

```
type Content_Access is access constant String;
```

and for the `lines` format it defines:

```
type Content_Array is
   array (Natural range <>) of access constant String;
type Content_Access is access constant Content_Array;
```

These type definitions give access to a readonly binary or string content and provides enough information to also indicate the size of that content. Then when the `--name-access` option is passed, the code generator declares and implements the following function:

```
function Get_Content (Name : String) return Content_Access;
```

That function will return either a content access or null if it was not found.

By default, when the `--content-only` option is not passed, the code generator provides more information about the embedded content such as the file name, the modification time of the file and the target file format. In that case, the following Ada record is declared in the Ada specification:

```
type Name_Access is access constant String;
type Format_Type is (FILE_RAW, FILE_GZIP);
type Content_Type is record
  Name    : Name_Access;
  Content : Content_Access;
  Modtime : Interfaces.C.long = 0;
  Format  : Format_Type := FILE_RAW;
end record;
```

The generated `Get_Content` function will return a `Content_Type`. You must compare the result with the `Null_Content` constant to check if the embedded file was found.

When the `--list-access` option is passed, the code generator emits a code that gives access to the list of file names embedded in the resource. The list of names is a simple Ada constant array. The array is sorted on the name. It is declared as follows:

```
type Name_Array is array (Natural range <>) of Name_Access;
Names : constant Name_Array;
```

The `map` format is special as it produces a mapping table defined by reading a JSON or XML content. Files matched by the rules are read and parsed to populate the internal map table. Content of these files is not available directly but the mapping is queried by using the generated `Get_Content` function with the name.

5.2 C Generator

The C code generator produces for each resource description a C header and a C source file with the name of that resource. The header contains the public declaration and the C source file contains the generated files with an optional function that allows to query and retrieve the file content. The C code generator is driven by the resource description and also by the tool options.

The header file declares a C structure that describes the content information. The C structure gives access to the content, its size, the modification time of the file and the target file format. The structure name is prefixed by the resource name.

```
struct <resource>_content {
```

```
  const unsigned char* content;
  size_t size;
  time_t modtime;
  int format;
}
```

This type definition gives access to a readonly binary content and provides enough information to also indicate the size of that content. Then when the `--name-access` option is passed, the code generator declares and implements the following function:

```
extern const struct <resource>_content *
     <resource>_get_content(const char* name);
```

That function will return either a pointer to the resource description or null if the name was not found.

When the `--list-access` option is passed, the C code generator also declares two global constant variables:

```
extern const char* const <resource>_names[];
static const int <resource>_names_length = NNN;
```

The generated array gives access to the list of file names embedded in the resource. That list is sorted on the name so that a dichotomic search can be used to find an entry.

5.3 Go Generator

The Go code generator produces for each resource description a Go source file with the name of that resource. The header contains the public declaration and the C source file contains the generated files with an optional function that allows to query and retrieve the file content. The C code generator is driven by the resource description and also by the tool options.

The Go source file declares a structure that describes the content information. The structure is declared public so that it is visible outside the Go package. It gives access to the content, its size, the modification time of the file and the target file format.

```
type Content struct {
  Content []byte
  Size    int64
  Modtime int64
  Format  int
}
```

This type definition gives access to a binary content and provides enough information to also indicate the size of that content. Then when the `--name-access` option is passed, the code generator declares and implements the following function:

```
func Get_content(name string) (*Content)
```

That function will return either a pointer to the resource description or null if the name was not found.

When the `--list-access` option is passed, the Go code generator makes available the list of names by making the `Names` variable public:

```
var Names= []string {
...
}
```

The generated array gives access to the list of file names embedded in the resource. That list is sorted on the name so that a dichotomic search can be used to find an entry.