astrium

generate

Text Generation Tool

generates text files or text components in database end items
based on a template and tables

Franz Kruse
Astrium Bremen, IO 62

2001-02-20

astrium

Table of Contents

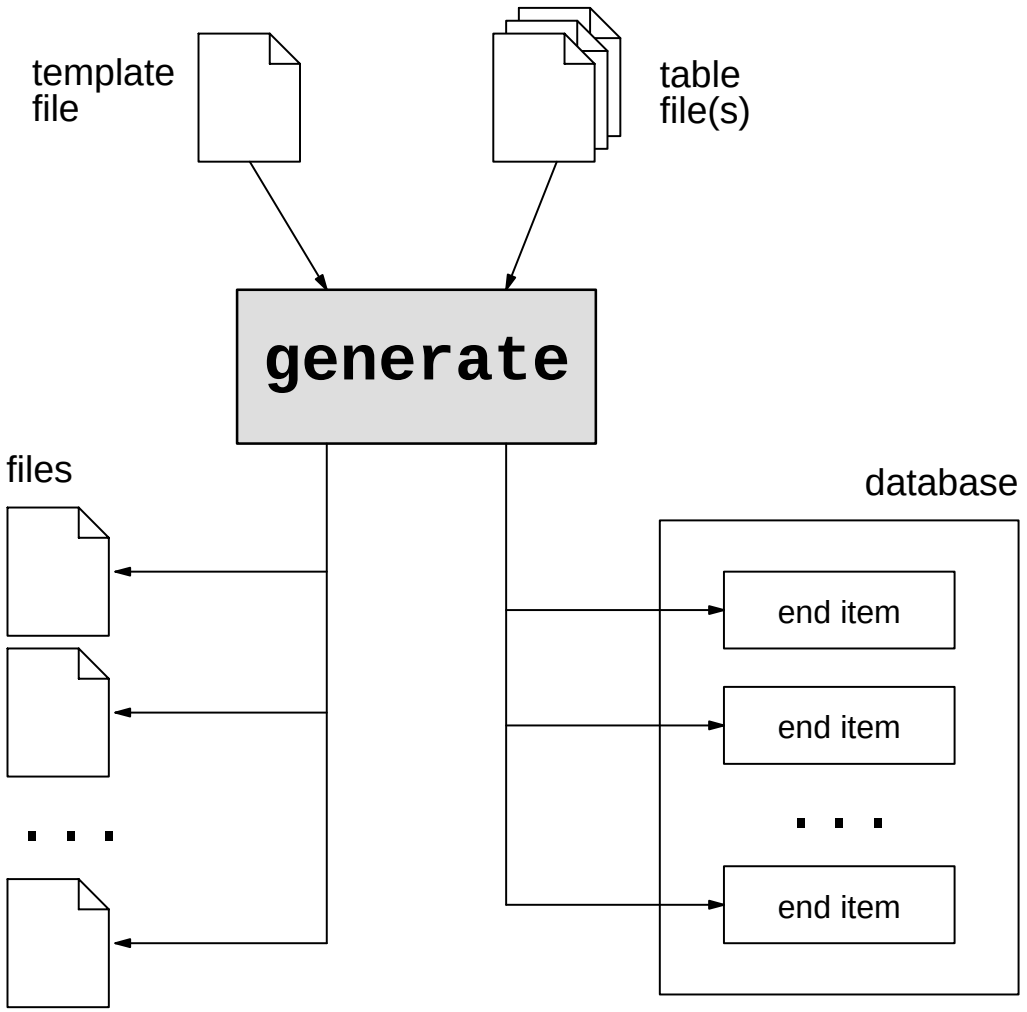
1 The generate Tool

The generate tool is a general purpose text generator that can produce a number of texts derived from a common template. The template is given as a text file with placeholders: macros defined in a table and parameters from the command line. Macro placeholders are denoted in the form <<NAME>> or <<NAME(parameters)>>, parameter placeholders in the form <<n>>, where n denotes the n–th parameter. As part of the generation process, placeholders are replaced with their current values.

The generated texts may, alternatively, be placed

- in files. For each generated text a separate output file may be generated, or all texts may be generated in one file.
- in the Columbus database. The generated text is then regarded as UCL/HLCL/CPL source code for one of the following item types. The target items must exist, they are not created by the program.
  - UCL automated procedures (APs)
  - HLCL command sequences
  - CPL scripts
  - any parameterized end items (the generated text is the formal parameter list definition).

The set of texts to be generated is defined in a table supplied as one or more text files. Each group of macro definitions generates a text from the template with the placeholders replaced by the current values of the macros and parameters. Optionally, all generated texts may be appended to just one file. This may be used to produce list, scripts, tables, etc. The value of the special built–in macro TARGET determines the name of the output file or end item.



2 Program Invocation

The program is invoked from the UNIX command line with the command

```
generate template_file table_file... options
```

Parameters:

`template_file`	name of the file containing the text template
`table_file...`	name(s) of the file(s) containing the macro table

Options:

Option names may be abbreviated by dropping characters from the end of the name, as long as the name remains unique. They may be written in lower, upper or mixed case.

`–environment` *`environment`*

defines the database user environment. If omitted, the environment will be obtained from the environment variable `MDA_ENVIRONMENT` in the same syntactic form:

```
<environment> ::=
  CCU <element_config> <mission> <system_tree_version> <ccu> |
  CDU <element_config> <mission> <system_tree_version> <cdu>

<ccu> ::=
  <pathname> <ccu_name> <version>.<issue>.<revision>

<cdu> ::=
  <pathname> <version>.<issue>.<revision> <test_version>
  <instance>
```

Example:

```
–environment ″CCU APM MASTER 4 \APM\FLTSYS FLAP_TEST 1.0.0″
```

`–parameters` *`value`* `...`

defines template parameters. Within the template, these parameters may be referenced in the form `<<n>>`, where `n` is a positive integer and refers to the n–th parameter. Each parameter is treated as a template that may contain macro placeholders. It is reevaluated each time the parameter is referenced, replacing the macros with their current value. For a description of macros see 4.2.

`–header` *`file_name`* `...`

name(s) of one or more header template files. These templates define text to be generated once before any other text. This is useful when generating a list (script etc.) that requires a header. The header templates may contain macro and parameter placeholders, they are evaluated right before the first text from the main template is generated. Placeholders are thus replaced with global macro/parameter values that are in effect at that time. Access to local parameters is not possible.

`–footer` *`file_name`* `...`

name(s) of one or more footer template files. These templates define text to be generated once after any other text. This is useful when generating a list (script etc.) that requires a footer. The footer templates may contain macro and parameter placeholders, they are evaluated after the first text from the main template has been generated. Placeholders are thus replaced with global macro/parameter values that are in effect at that time. Access to local parameters is not possible.

`–target [value]`

This option overwrites the value of the `TARGET` macro, which defines the name of the output file or database end item. value is treated as a template that may contain macro placeholders, it is reevaluated each time a text is to be generated, replacing the macros with their current value. If `value` is omitted, the texts are generated to standard output. For a description of the `TARGET` macro see 4.2.4.

`–onboard_view`

instructs the program to use the onboard view in the database, i. e. it sets the onboard flag for APs, and it references all end items with their unmapped original item type.

`–strict`

require strict macro checking. Normally a reference to a macro that is not defined evaluates to an empty string. When the `–strict` option is given, however, such references yield an error message.

`–abort`

instructs the program to abort in case of errors. The standard behaviour is to continue text generation.

`–check`

When this option is given, the program performs its generation process, but the actual output is suppressed. This can be used to check the template(s) and table(s) for correctness.

`–verbose`

When this option is given, the program lists all generated (or checked, resp.) file or end item names.

Examples of programm calls

`generate template.txt table.tab`

Generate texts from the template contained in file `template.txt`, take makro definitions from the file `table.tab`. The MDA environment information is in variable `MDA_ENVIRONMENT`.

`generate template.txt table1.tab table2.tab –verbose`

Generate texts from the template contained in file `template.txt`, take makro definitions from the concatenated files `table1.tab` and `table2.tab`. Output a list of all generated or checked items.

`generate template.txt table1.tab table2.tab –verbose –check`

Same, but don’t actually generate texts. Just check the input files.

`generate template.txt table.tab –h header.txt –f footer.txt –target`

Same as first example, but generate a header (whose template is in file `header.txt`) and a footer (whose template is in file `footer.txt`). Output the generated texts on standard output.

`generate template.txt table.tab –par abc xyz`

Same as first example. The template may contain references to the parameters `abc` and `xyz`. The reference `<<1>>` will be replaced by `abc`, the reference `<<2>>` by `xyz`.

3 Templates

A template is a text pattern from which a number of final texts are to be generated. It contains placeholders which will be replaced by actual values taken from the table file(s). All other text will simply be copied. Placeholders can have any of the following forms, for a detailed description of macros see 4.2

`<<NAME>>`

a single macro call without parameters. The macro will be replaced by its value as defined in the table file.

`<<NAME (parameter, parameter, ..., parameter)>>`

a macro call with parameters. The macro will be replaced by its value as defined in the table file. The macro value may contain parameter references of the form `<<n>>`, where n is the number of the parameter, starting at 1. Each reference `<<n>>` will be replaced by the n–th parameter from the macro call. Parameter references for which there is no actual parameter in the call will be replaced by an empty string. The parameters may themselves contain placeholders which will be replaced by values, before the parameters are inserted into the macro value.

`<<"text">>`

a macro that will be replaced by the text enclosed in quotes. This pseudo–macro can be used to insert text that would otherwise be interpreted as meta characters (e. g. placeholder symbols) or comments, or blanks that would otherwise be removed.

`<<?condition>>then_branch<<|>>else_branch<<.>>`

a conditional expression that will be replaced either by the `then_branch` template or the `else_branch` template, depending on a logical expression formed by placeholders.

`<<n>>`

a reference to a parameter given with the –parameters option in the command line. Each reference `<<n>>` will be replaced by the n–th parameter from the `–parameters` option. References for which there is no actual parameter in the `–parameters` option will be replaced by an empty string.

A program call may involve three templates. All three templates may contain placeholders, as described above.

- the *main template*

  This is the template for for the repeated texts to be generated. It must always be supplied.

- the *header template*

  From this template a header text is generated once before any other text. The macro values inserted are those that are in effect for the first text from the main template. Only global macros can be referenced, local macros are not visible. The header template is optional.

- the *footer template*

  From this template a footer text is generated once after all other texts have been generated. The macro values inserted are those that are in effect for the last text from the main template. Only global macros can be referenced, local macros are not visible. The footer template is optional.

4 Tables and Macros

4.1 Table Layout

The second parameter to the `generate` program, the table file(s), defines which texts are to be generated with which values inserted, and where the generated texts are to be written. If several table files are given, they are simply treated as one big file concatenated from the single files. This allows to split up a table in smaller pieces for structuring purposes.

A table is a list of macro definitions. Macros fall in two classes: *global* macros and *local* macros. Global macros keep their value through several generated texts until they are explicitly reassigned. Local macros have their values only for one generated text. As soon as a text has been generated, they lose their values and must be reassigned for the next text. Global and local macros may, howenver, reference each other arbitrarily. Global macros are always defined in *keyword notation* in the form

```
NAME : value
```

The definition must be written at the left margin, without any leading spaces or tabs.

The local macro definitions are made up in repeated groups, and each group generates a text from the template with macro calls replaced by values defined in this group. The layout of the table allows two alternative forms to define and group local macros: *keyword notation* and *column notation*:

4.1.1 Keyword Notation

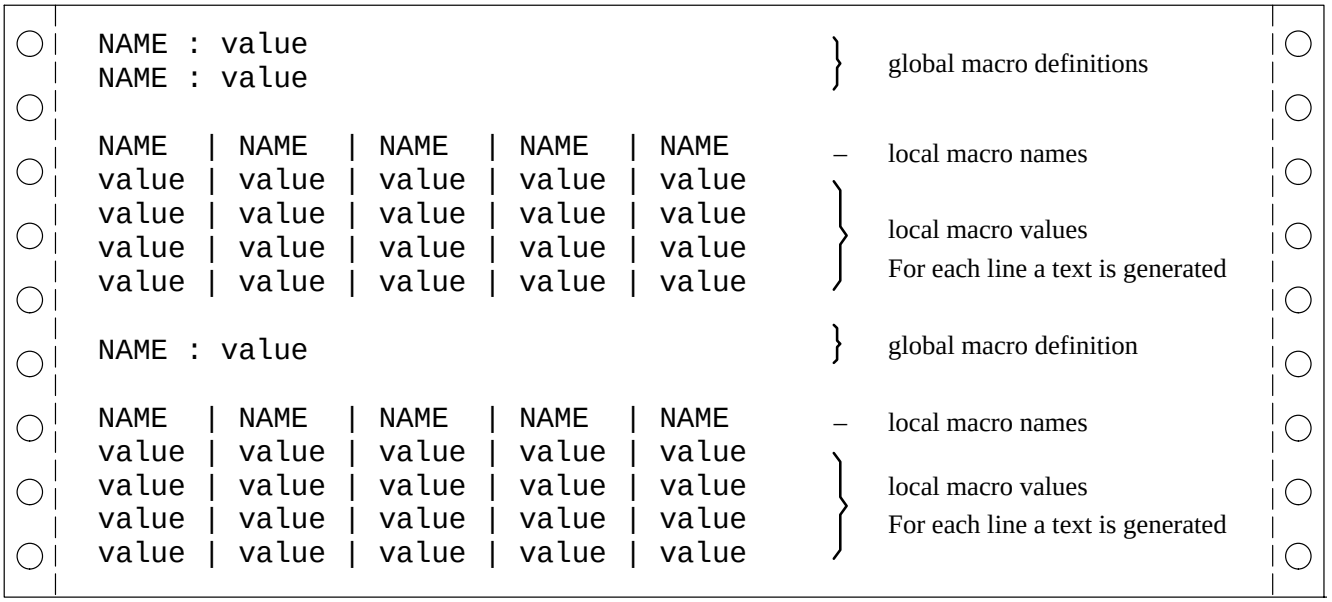
The form is like for global macros, but at least one blank (space or tab character) must precede the definition. This is what syntactically distinguishes local from global macro definitions. Groups of local macros are separated by an empty line. A table file in keyword notation may thus look like this:

```
NAME : value          } global macro definitions
NAME : value

  NAME : value        }
  NAME : value        } group of local macro definitions
  NAME : value        } This group generates one text.
  NAME : value        }

  NAME : value        }
  NAME : value        } group of local macro definitions
  NAME : value        } This group generates one text.
  name : value        }

  ...

NAME : value          } global macro definition

  NAME : value        }
  NAME : value        } group of local macro definitions
  NAME : value        } This group generates one text.
  NAME : value        }

  ...
```

4.1.2 Column Notation

In column notation, macro values are written in columns separated by vertical bars, like in a table, each column representing a macro. The first line of each column must be the name of the corresponding macro. Whenever the table is interrupted by a keyword definition (e. g. a global macro definition), a new table begins, and the first line must again contain the macro names. Each line of the table represents a group of local macros that generates a text.

A table can only hold local macro definitions. It need not be indented with blanks. A table file in column notation may look like this:



4.1.3 General Rules

For both table forms, the following rules apply

- Macro names must start with a letter that may be followed by any combination of letters, digits, underscore characters and dots. Macro names are not case sensitive.
- The macro values must be written in exactly the form they are to be represented in the generated text. Note that leading and trailing blanks (spaces and tab characters) are removed from the value, internal blanks are kept.
- Blanks (spaces and tab characters) may be freely inserted between any items.
- Empty lines may be freely inserted. But note, that within a group of local macros (or a table) an empty line designates the end of the group.
- Comments may be appended to any (empty and non–empty) line. A comment starts with two hyphens (`--`) and extends to the end of the line.
- Lines shown in the above pictures must be written in one line. The separator `<<>>` can be used to break up a logical line into several physical lines. For a description of separators see 6.

4.1.4 Target Files and Pathnames

The built–in global macro `TARGET` is evaluated for each text to be generated. Its current value determines the file or database item where the text is to be generated. If it starts with a backslash (`\`), it is assumed to be a database pathname, otherwise a file name. If it is an empty string or undefined, the text is generated on standard output.

Like other macros, the value of the `TARGET` macro may contain calls to other (global and local) macros. So one global definition may be sufficient to set its value differently for each text to be generated. As long as the value does not change, all texts are written into the same file or database item. This may be used to produce lists.

4.2 Macros

4.2.1 Text Macros

The value of a text macro is a piece of text that replaces the macro placeholder when text is generated from a template. The value of a text macro may contain parameter references, these will be replaced by the actual parameters passed in calls to this macro. It may also contain macro calls with or without parameters. When the macro is being expanded, all parameter references and macro calls are first evaluated and expanded. In this way, macro calls may be cascaded at any depth, but circular references are not allowed.

Example: Given these macro definitions

```
FILE_NAME : <<DIRECTORY>>/<<1>>.<<SUFFIX>>
DIRECTORY : /user/columbus/users1/<<PROJECT>>
PROJECT   : flap1
SUFFIX    : dat
```

the macro call

```
FILE_NAME(xyz)
```

will be expanded as

```
/usr/columbus/users1/flap1/xyz.dat
```

4.2.2 Counter Macros

A counter macro call generates a number, and each call increments the number by 1. There is a built–in counter macro named # (called in the form <<#>>) with the default initial value 1. Successive calls will generate the sequence

```
1 2 3 4 ...
```

Users may define their own counter macros, specifying the starting value in the form

```
#NAME : value
```

where value must be an integer (both signed and unsigned) and is the first value generated when calling the macro in the form <<#NAME>>. Like text macros, counter macros can be global or local. Global counters keep their value across generated texts, local macros are reset to their initial value for each generated text. The built–in counter # is global, it may be redefined as

```
#COUNTER : 10
```

User defined counter macros may be called in two forms: <<#NAME>> and <<NAME>>. The first form yields an incremented value with each call, the second form does not increment, it just repeats the last value. The value is undefined, if the macro has not been called before. For the definition

```
#COUNTER : 10
```

the template

```
<<#COUNTER>> – <<#COUNTER>> – <<COUNTER>> – <<COUNTER>> – <<#COUNTER>>
```

will generate

```
10 – 11 – 11 – 11 – 12
```

All counter macros have two optional parameters: the minimum width and a filler character, which, by default, is a space. The template

```
<<#COUNTER>> – <<#COUNTER(5,0)>> – <<COUNTER(5)>> – <<COUNTER(5,*)>>
```

generates

```
10 – 00011 –    11 – ***11
```

4.2.3 Identity Macro <<"text">>

A pseudo–macro of the form <<"text">> can be used to insert an immediate text. The text must be enclosed in quotes. This can be used to insert text that would otherwise be interpreted as meta characters (e. g. as placeholder syntax) or comment, or blanks that would otherwise be removed from the start or end of a macro definition.

4.2.4 Built–in Macros

The following macros are built–in:

`TARGET`

The value of this macro determines the target location for the text to be generated, i. e. the file or database item where the text is to be generated. If it starts with a backslash (\), it is assumed to be a database pathname, otherwise a file name. If it is an empty string or undefined, the text is generated on standard output. This macro is global. It is evaluated for each text to be generated.

Example:

```
TARGET : <<DIRECTORY>>/<<FILE>>.<<SUFFIX>>
```

This target specification will generate file names depending on the current value of the referenced components (directory, file name and suffix). These may be local macros that change for each text to be generated. As long as the value does not change, all texts will be written in the same file or database item. This may be used to generate lists.

If the command line option `–target` is given, the `TARGET` macro is ignored, and the option value is used instead. Note that the option value may contain macro calls as well.

`DATE`

This macro generates the current date in the form `dd.mm.yyyy`. Assignments to the macro are not allowed.

`TIME`

This macro generates the current time in the form `hh:mm:ss`. Assignments to the macro are not allowed.

`CHECK (name [, item_type])`

This macro always generates an empty string. Only its side–effect is relevant: it checks for the existence of a file or database item, whose file name or pathname must be given as a parameter. If the name starts with a backslash (\), it is assumed to be a pathname, otherwise a file name. For pathnames the required item type may be passed as a second parameter. If given, it is not only checked that the database item exists, but also that is has the specified item type. Assignments to the macro are not allowed.

`# [(width [, filler_char])`

This is a counter macro. Each call yields a number incremented by 1. The first generated number is 1 by default. It may be changed by an assignment:

```
# : value
```

where `value` must be an integer (positive, negative or 0). The assigned number will then be generated by the next call.

Like all counter macros, # may have one or two parameters: a minimum width and a filler character. If given, the generated number will occupy at least `width` characters in the generated text, and filler characters will be inserted on the left, if needed. The default filler character is a space.

5 Conditional Placeholders

Placeholder replacement can be made dependent on conditions. A conditional placeholder has one of the forms

```
<<?condition>>then_branch<<.>>
<<?condition>>then_branch<<|>>else_branch<<.>>
```

The single parts have the following meaning:

`<<?condition>>`

This is the condition that evaluates to true or false. Currently `condition` can only be a placeholder (macro call with or without parameters, or a parameter reference). If the placeholder evalutates to a non–empty text, the condition is considered true, otherwise false. Future extensions may provide for more complex expressions.

`then_branch`

This is a template that will be selected if the condition is true.

`else_branch`

This is a template that will be selected if the condition is false.

`<<|>>`

a separator that syntactically separates the two alternative branches.

`<<.>>`

a separator that syntactically terminates the conditional placeholder.

The two branches are normal templates, i. e. they may again contain any kind of placeholders. A consequence is that conditional placeholders can be nested, theoretically at any depth.

Example:

```
<<?PREFIX>><<PREFIX>>-<<BODY>><<|>><<BODY>><<.>>
```

6 Separators

Some special separators have a purely syntactical function:

`<<|>>`

branch separator, used to separates alternative branches, e. g. in conditional placeholders.

`<<.>>`

termination separator, used to mark the syntactical end of constructs, e. g. in conditional placeholders.

`<<>>` (currently not implemented)

line concatenator, used to join two physical lines into one logical line. It allows to split one logical line in several physical lines for formatting purposes. After `<<>>` only blanks and comments are allowed. The logical line containing the separator is continued with the first non–blank character from the next phisical line.

7 An Example

The following is a somewhat complex example taken from the Flight Automated Procedures (Flight APs, or FLAPs) development for the Columbus Orbital Facility (COF). It shows how the generation tool can be used for a number of tasks involved in a software development process. The example has been simplified, in order to keep it easily understandable.

Automated procedures in the Columbus project are written in the User Control Language (UCL). Compilation units are kept and maintained in the mission database. The UCL compiler retrieves the source code from the database and stores the object code and other generated data back in the database.

A large subset of flight APs perform basically the same activity, applied to different on–board hardware devices: they send a command to switch the equipment in a specific mode, then wait for a certain amount of time, read a corresponding measurement and check it for the expected effect of the command, and issue an event message reporting success or failure.

Groups of APs of such classes can be generated from a common source code template. This is the simplified template for the group of APs handling discrete commands. Placeholders are shown in bold face:

File: `discrete.tmpl`:

```
-------------------------------------------------------------------------------
--  <<AP_PATH>>
--
--  Issue command     <<COMMAND_PATH>>
--  Check measurement <<MEASUREMENT_PATH>>
-------------------------------------------------------------------------------
-- Date     Author              Change description
-------------------------------------------------------------------------------
-- <<DATE>> Autom. generation   Initial version
-------------------------------------------------------------------------------

procedure <<AP_NAME>>;

  import \APM\INT\ENV\OB_UCL\ONBOARD_LIBRARY;
  import \APM\FLTSYS\APP_SW\FLAP\USER_LIB\PATH_NAME_ALIAS;

  variable Status       : UCL_RETURN;
  variable Sensor_Value : statecode;

begin
  ISSUE_DISCRETE (COMMAND : <<COMMAND_PATH>>,   -- send the command
                  ...
                  STATUS  : Status);

  DELAY (<<DELAY>>);                            -- wait for effect

  Sensor_Value := <<MEASUREMENT_PATH>>;         -- read the measurement

  if <<CONDITION (Sensor_Value)>> then          -- and check it
    SUBMIT_EVENT (EVENT_NAME : CREW_LOG_CMD_SUCCESSFUL (),  -- report success
                  STATUS     : Msg_Return_Status);

    halt SUCCESS;  -- return to calling AP with success indication

  else  -- indicate failure
    SUBMIT_EVENT (EVENT_NAME : CREW_LOG_CMD_FAILED (Command,  -- report failure
                                                    Measurement,
                                                    Sensor_Value),
                  STATUS     : Msg_Return_Status);

    halt FAILURE;  -- return to calling AP with failure indication
  end if;

end <<AP_NAME>>;
```

The Table

The table that defines the set of flight APs to be generated is split up in subtables for convenience: a header table that holds all global macro definitions and is used with all AP tables, and the actual AP tables that list the APs of the various classes. The tables shown here have been simplified for better understandability.

Header table

File: global.tab

```
-------------------------------------------------------------------------------
-- This table holds global macro definitions used for all FLAP tables.
-------------------------------------------------------------------------------

SW_TREE          : \APM\FLTSYS\APP_SW\FLAP                -- software name tree
HW_TREE          : \APM\FLTSYS\                           -- hardware name tree

BRANCH           : <<SUBSYS>>\<<UNIT>>\<<SEGMENT>>        -- branch in name tree

NAME_PREFIX      : <<?SEGMENT>><<>>                       -- prefix for AP name
                     <<UNIT>>_<<SEGMENT>><<>>
                   <<|>><<>>
                     <<UNIT>><<>>
                   <<.>>

AP_NAME          : <<NAME_PREFIX>>_<<COMMAND>>            -- UCL name of AP

AP_PATH          : <<SW_TREE>>\<<BRANCH>>\<<COMMAND>>     -- pathname of AP
COMMAND_PATH     : <<HW_TREE>>\<<BRANCH>>\<<COMMAND>>     -- pathname of command
MEASUREMENT_PATH : <<HW_TREE>>\<<BRANCH>>\<<MEASUREMENT>> -- pathname of measurem.

#SID             : 1000                                   -- start of SID range
```

FLAP tables (one of the FLAP tables for the various FLAP classes)

File: epds.tab

```
-------------------------------------------------------------------------------
-- This is the FLAP table for the EPDS subsystem.
-------------------------------------------------------------------------------

SUBSYS | UNIT | SEGMENT          | COMMAND | MEASUREMENT | CONDITION     | DELAY
EPDS   | PDU1 | A1_MAIN_PWR_BUS | ON2     | ONOFF_STS2  | <<1>> = $ON   | 0.5 [s]
EPDS   | PDU1 | A2_MAIN_PWR_BUS | ON2     | ONOFF_STS2  | <<1>> = $ON   | 0.5 [s]
EPDS   | PDU1 | A3_MAIN_PWR_BUS | ON2     | ONOFF_STS2  | <<1>> = $ON   | 0.5 [s]
EPDS   | PDU1 | A4_MAIN_PWR_BUS | ON2     | ONOFF_STS2  | <<1>> = $ON   | 0.5 [s]
EPDS   | PDU1 | O1_MAIN_PWR_BUS | ON2     | ONOFF_STS2  | <<1>> = $ON   | 0.5 [s]
...
EPDS   | PDU2 | F1_MAIN_PWR_BUS | OFF2    | ONOFF_STS2  | <<1>> = $OFF  | 0.5 [s]
EPDS   | PDU2 | F2_MAIN_PWR_BUS | OFF2    | ONOFF_STS2  | <<1>> = $OFF  | 0.5 [s]
EPDS   | PDU2 | F3_MAIN_PWR_BUS | OFF2    | ONOFF_STS2  | <<1>> = $OFF  | 0.5 [s]
EPDS   | PDU2 | F4_MAIN_PWR_BUS | OFF2    | ONOFF_STS2  | <<1>> = $OFF  | 0.5 [s]
EPDS   | PDU2 | O2_MAIN_PWR_BUS | OFF2    | ONOFF_STS2  | <<1>> = $OFF  | 0.5 [s]
...
```

Generation of the APs

The following command generates all of the APs from the template and the tables:

```
generate discrete.tmpl global.tab epds.tab –target "<<AP_PATH>>"
```

Other Tasks

Using the same central tables, a number of related tasks may be performed with the text generator, just be using different templates.

Check for all used commands and measurements

File: check_epds.tmpl

```
<<CHECK (COMMAND_PATH,     SPC_1553B_DIGITAL_INPUT)>>
<<CHECK (MEASUREMENT_PATH, SPC_1553B_DIGITAL_OUTPUT)>>
```

```
generate check_epds.tmpl global.tab epds.tab –check
```

Generate a shell script for compilation of all generated APs

File: compile_epds.tmpl

```
echo “Compiling <<AP_PATH>>”
$CLS_HOME/bin/sun5/cls_editor “<<AP_PATH>>”
```

```
generate compile.tmpl global.tab epds.tab –target compile_epds
source compile_epds
```

Generate input for the MDB Batch Data Entry tool

This requires a list of pathnames to be generated, together with their item type.

File: bde_epds.tmpl

```
<<AP_PATH>> UCL_AUTOMATED_PROCEDURE
```

```
generate bde_epds.tmpl global.tab epds.tab –target bde_epds_input.txt
```

Generate input for the emulated database

This requires a list of pathnames to be generated, together with their SID and item type.

File: emu_epds.tmpl

```
<<AP_PATH>> <<#SID>> UCL_AUTOMATED_PROCEDURE
```

```
generate emu_epds.tmpl global.tab epds.tab –target emu_epds_input.txt
```