Runtime Configuration for MPC - Developper Manual

Sogeti High Tech

September 11, 2013

Contents

1 Introduction

Since MPC 2.4.1, a configuration system has been introduced through the module `MPC_Config`: it enables the user to setup some parameters at the runtime when running his binary with MPC.

This manual will explain to a developper how the configuration system is designed and how to add parameter into it.

2 The module MPC_Config

All the configuration system is implemented into the `MPC_Config` module which gives functionnalities to generate:

- The configuration structure;
- The UNIX man for options list and values;
- The parsing source code;
- The displaying function.

This module also contains the graphical editor sources.

3 Sources of the modules MPC_Config

The module `MPC_Config` is subdivided into several directories:

- `bin`: contains the sources of the `mpc_print_config` executable;
- `doc`: contains the user and developper manuals describing the configuration system;
- `editor`: contains the graphical configuration editor;
- `generated`: contains all the generated files needed at the runtime (see §3.2 for more details);
- `generators`: contains all XSLT files used to generate the files in the `generated` folder;
- `src`: contains the sources for the configuration parsor from XML files (see §3.3 for more details).

3.1 Details for the editor folder

The `editor` folder contains:

- `config`: contains a set of different configs (valid, unvalid and malformed) which can be used as example for the editor;
- `images`: contains all the images used in the editor;
- `javascript`: contains all the JavaScript scripts used to load and edit configuration files;
- `style`: contains all the CSS files defining the editor style;
- `index.html`: the main file to open in a browser to use the graphical editor.

3.2 Details for the generated folder

Several files are generated using the XSLT in the `generators` folder:

- `sctk_runtime_config_struct.h`: define all the C data structures (`struct`, `enum`, etc.) of the MPC configuration structure;
- `sctk_runtime_config_struct_meta.c`: define meta-description (datatype, offset into the structure, etc.) to load the MPC configuration structure;
- `sctk_runtime_config_struct_defaults.h`: define functions prototypes initializing the MPC configuration structure with the default values;
- `sctk_runtime_config_struct_defaults.c`: initialize the MPC configuration structure with the default values;
- `global-config-meta.xml`: contains the contents of each `config-meta.xml` existing in MPC;
- `mpc_config.5`: UNIX man describing all the parameters (type, default value, doc) of the MPC configuration structure;
- `mpc-config.xsd`: scheme to validate the final configuration file.

3.3 Details for the src folder

The `src` folder contains the following files:

- `sctk_runtime_config_mapper.{.h,.c}`: provide the functions to convert the XML configuration file to the C structure;
- `sctk_runtime_config_printer.{.h,.c}`: use by the `mpc_print_config` executable to display the parameters for the XML configuration file, using the file `sctk_runtime_config_struct_meta.c`;
- `sctk_runtime_config_selectors.{.h,.c}`: handle selectors to select dynamically profiles at execution time;

- `sctk_runtime_config_sources.{.h,.c}`: provide the functions to open XML configuration files and to select profiles to apply;
- `sctk_runtime_config_validation.{.h,.c}`: provide a function to overwrite parameters with environment variables, and a function to check the values of the parameters;
- `sctk_runtime_config_walk.{.h,.c}`: use to run over the C configruation structure in order to display its contents;
- `sctk_runtime_config.{.h,.c}`: provide the interface that will be used in the other MPC modules.

A module `sctk_libxml_helper.{.h, .c}` is also developped to use `libxml2` to read and write XML files.

4 The configuration system into MPC

4.1 Write a `config-meta.xml` file

Each module to be added in the configuration system needs its own `config-meta.xml` file. The general structure of such a file is designed as follow:

General structure of a `config-meta.xml` file

```
<config>
  <usertypes>...</usertypes>
  <modules>...</modules>
<profile>
```

The developer has to defined all the types (`structure`, `union`, etc.) that will be used into the configuration system in the `usertypes` section.

4.1.1 The `usertypes` section

In this section, the developer will define structures (`struct`), unions (`union`) and enumerators (`enum`) that will be used in the configuration system. Each one of them have two required attributes: `name` (resp. `doc`) is the name (resp. the description) of the variable/object.

A structure (as a C/C++ struct) is a set of `param` or `array` which have required attributes:

1. `name` which is the name of the variable;
2. `type` which is its type (simple: `int`, `float`, ... or complex: `struct`, `union`, ...),
3. `doc` which shortly describes what it is for,
4. `[`only for `array]` `entry-name` which matches to the tag name of the array elements.

The `param` have an optional attribute, `default`, to initialize the variable to a specific default value.

Example of a `struct` definition

```
<struct name="my_struct" doc="This␣a␣struct␣test">
  <param name="param1" type="string" doc="String␣param." />
  <param name="param2" type="user_type" doc="Param␣of␣type␣user_type␣(enum,␣struct,␣...)." />
  <array name="array1" entry-name="elt" type="int" doc="Array␣of␣int␣with␣elt␣as␣xml␣tag␣name." />
</struct>
```

An union (as a C/C++ union) matches to a list of choices defined by a name and a type.

Example of a `union` definition

```
<union name="my_union" doc="This␣an␣union␣test">
  <!-- Define a choice with a string param -->
  <choice name="choice1" type="string" />
  <!-- Define a choice with an user type param -->
  <choice name="choice2" type="user_type" />
  <!-- Define a choice with an int param -->
  <choice name="choice3" type="int" />
</union>
```

An enumerator (as a C/C++ enum) is a set of possible values as shown in the following code.

Example of a `enum` definition

```
<enum name="day" doc="Week days">
  <value>Monday</value>
  <value>Tuesday</value>
  <value>Wednesday</value>
</enum>
```

4.1.2 The `modules` section

In this section, the developer will declare the high-level structures he wants the user to configure. In the following code, there will be two configurable structures of type `my_struct_1` and `my_struct_2`.

Example of a `modules` definition

```
<modules>
  <module name="struct_1" type="my_struct_1" />
  <module name="struct_2" type="my_struct_2" />
</modules>
```

4.2 Configuration management workflow

The workflow of configuration management can be described by steps:

1. Write a `config-meta.xml` for each MPC module which needs to be integrated into the configuration system;
2. Run `mpc_gen_runtime_config` which will :
   - Aggregate all the `config-meta.xml` to generate the `global-config-meta.xml`;
   - Apply XSLT transformations to generate source code for configuration management;
3. Compile MPC;

The Figure 1 summarized this process.

4.3 Steps for developper

A developper who wants to integrate his MPC module into the configuration system must:

1. Create a configuration file `config-meta.xml` in his module and define all the options he wants to parametrized;
2. Mark the dependency to the MPC_Config module by adding in the file `module_dep`: `need_module MPC_Config`;
3. Regenerate the MPC_Config auto-generated files by execution `./MPC_Tools/mpc_gen_runtime_config` from `mpc` directory;
4. Include the header `sctk_runtime_config.h` in his source files;
5. Use the function `sctk_runtime_config_get()` to access to the configuration structure.

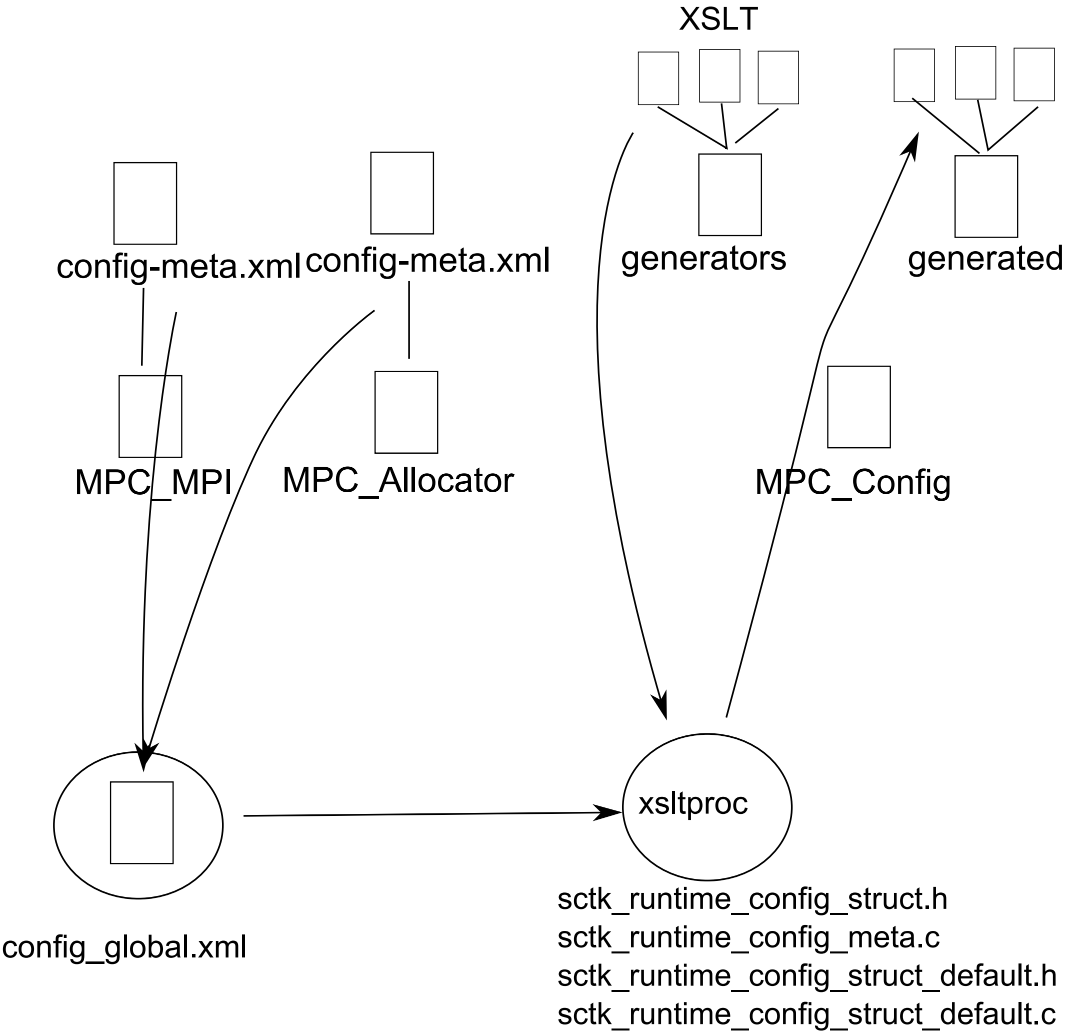
XSLT
config-meta.xml
config-meta.xml
generators
generated
MPC_MPI
MPC_Allocator
MPC_Config
xsltproc
config_global.xml
sctk_runtime_config_struct.h
sctk_runtime_config_meta.c
sctk_runtime_config_struct_default.h
sctk_runtime_config_struct_default.c

Figure 1: Representation of the workflow

5 The graphical editor interface

To edit the generated configuration files, an editor has been developed in HTML/JavaScript. The development has been validated on Firefox 10.0.8 ESR, but works with several others browsers as Google Chrome or Safari.

This section will only describe the JavaScript sources. In order to make future evolutions easy, the code is organized in a Model-View-Controller pattern. The model represents the data read from the XML configuration file: it is an association of hash tables which store propeties informations (name, type, value(s), etc.). The view is the graphical representation of the model, and the controller manages the events to synchronize both the model and the view. Some extra libraries have been included:

- `EditorXML.js` to manage the XML: conversion functions, getting child nodes, etc.;
- `xmllint.js` is a module compiled from libxml, which is used to validated the XML files (data and structure speaking);
- `BlobBuilder.js` and `FileSaver.js` to handle the saving of the new XML generated file.

5.1 The model

The model is handled in the `EditorModel.js` which can be separated in several parts:

- The functions used to create the model from an XML file;
- The functions used to update the model;
- The functions used to generate an XML file from the model.

A configuration file is divided in three parts, and each one has its own generation function:

1. The `profiles` section is generated by `createProfilesModel(config);`
2. The `networks` section is generated by `createNetworksModel(config);`
3. The `mappings` section is generated by `createMappingsModel(config).`

where `config` matches to the XML read configuration file converted into an DOM Document HTML Object.

The model generation has been thinked in such a way that it is fully automatic. While the XML configuration file is reading, the functions reads in the `sctk_runtime_config_meta.js` (which is a conversion of `sctk_runtime_config_meta.h` generated with XSL in JavaScript format) to get the type of each property. Depending on its type (`param`, `array`, `struct`, `union`), the model of a property is generated with one of the following functions:

- `createStructModel(xml, js);`
- `createArrayModel(xml, js);`
- `createParamModel(xml, js);`
- `createUnionModel(xml, js);`

where `xml` matches to the property read in the XML configuration file and `js` is its meta-information.

If a new type called `XXX` is inserted in the configuration system, the developer just has to developed a function `create`*XXX*`Model(xml, js)` to create the associated model for a given property.

`EditorModel.js` also provides functions to update the model when a property value has been changed in the editor. Here is a list of some available functions:

- `updateModel` to update a module value in a profile;
- `upateMappingsModel` to update the mappings section;
- etc.

Finally, this file contains the XML generation functions. Each part of the configuration file has its own, and each type too. In this way, the XML generation is fully automatic:

- `generateMappingsXmlConfig` to generate the XML code for the mappings section;
- `generateStructXML` for a property of `struct` type;
- `generateParamXML` for a property of `param` type;
- etc.

5.2 The view

The view is handled in the `EditorView.js` which is also divided in several parts:

- The functions used to create the view from the model;
- The functions used to update the view when the model is updated.

As for the model, a generation function is developed for each part of a configuration file:

1. The `profiles` section is generated by `createProfilesView`;
2. The `networks` section is generated by `createNetworksView`;

3. The `mappings` section is generated by `createMappingsView`.

Each property type has its own function to create its representation:

- Numbers, function pointers and strings will be drawn an input text;
- Enumerators are represented with combo box;
- Sizes are an association between an input text and a combo box.

5.3 The controller

The file `EditorController.js` gives functionnalities to manage editor events to synchronize the model and the view.

- Adding/deleting new elements (profiles, mappings, etc.);
- Property updating;
- etc.

5.4 The XML handling

The file `EditorXML.js` implements some functions to manage XML files.

The `StringToXML` (resp. `XMLToString`) converts an XML document into a string (resp. a string to an XML document). `formatXML` formats a XML code to respect indentation tags.

XML parsing functions are also implemented:

- `getNodes` gets all the XML elements matching to a given name;
- `hasChild` returns the child of a given XML element;
- `getChildNodes` returns all the child of a given XML element;
- `getNodeValue` returns the text value of a given XML element.

To ensure the portability of the editor, the `getElementsByClassName` has been recoded: it returns all the XML elements of a given node (the entire document by default) matching to a class name.

5.5 Opening and saving local files

JavaScript forbids to open and save local files for security reasons.

However, the module `FileReader` (new in HTML5) lets the user to open local files. Events are associated to this class:

- `onload` is triggered at the end of the file reading. Once the content loaded, it is analyzed by `xmllint.js` and the associated XSD file to validate its consistency. If the XML file is not valid, it will not be opened, and an error message will be displayed. Otherwise, the file will be loaded.
- `onerror` is triggered when an error occurs during the file reading. If this event is catched, the XML file will not be loaded.

To save files on a local disk storage, classes `BlobBuilder.js` and `FileSaver.js` have been added to the editor.