Run-Time I/O Layer and Auto-Documentation of Users Guide: Requirements and Design

MPAS Development Team

March 22, 2013

Contents

Chapter 1

Summary

This document describes the requirements and design of two projects that share implementation details.

The first of these two projects is creating a run-time configurable I/O layer. This I/O layer will be used to add and remove fields from I/O streams at run-time rather than at build time.

The second project is the creation of a standardized Registry format that allows the documentation of variables, namelist options, dimensions, and streams to all be housed under the Registry files.

Details related to both of these will be in the coming sections.

Chapter 2

Requirements

This chapter lays out the requirements for both projects. From the requirements one can see the overlap between the two.

2.1 Run-time I/O Layer

2.1.1 Requirement: Variables can be added/removed from I/O streams at run-time

Date last modified: 02/15/13
Contributors: (Doug Jacobsen)

The run-time configurable I/O layer should provide the ability for modification of I/O streams at run time, as opposed to the current structure where it's at compile time.

2.1.2 Requirement: Ability to create new streams at build time

Date last modified: 02/15/13
Contributors: (Doug Jacobsen)

The new I/O layer should make the creation of a new stream available to a developer/user. Although it would be preferred to have stream creation possible at run-time, it's currently difficult to see an easy way to do this.

This requirement also means modifying Registry to create streams, which in turn means defining a standard for stream definitions.

2.1.3 Requirement: Ease of use for users

Date last modified: 02/15/13
Contributors: (Doug Jacobsen)

The format of the I/O layer configuration files should be easy to use for developers/users of the MPAS framework.

2.2 Auto Documentation

2.2.1 Requirement: CF Compliant Output

Date last modified: 02/15/13
Contributors: (Doug Jacobsen)

Although Registry makes adding and removing fields easy currently, it's difficult to define attributes that should be added to an output file or field.

This requirement would make us re-evaluate how Registry looks to users and works behind to scenes to make CF compliance an easier thing to achieve.

2.2.2 Requirement: Built-in Documentation

Date last modified: 02/15/13
Contributors: (Doug Jacobsen)

Registry should be modified with the ability to contain additional information for namelist options, dimensions, variables, and streams. These should include things like units, descriptions, possible values, etc.

2.2.3 Requirement: Auto-documentation generation parser

Date last modified: 02/15/13
Contributors: (Doug Jacobsen)

While documentation is something we strive to improve, it's currently a daunting task. There is an issue with maintaining documentation, especially of Registry entries, in a way that makes it clear to a user what they are.

Eventually this information should be placed within a users guide for each core, however that task alone is rather large in itself. This requirement would make available another parser of the Registry that, given the standardized format and additional documentation, would create the LaTeX documentation to be included in a users guide. This would make maintaining of that portion of the users guide to be significantly easier.

Chapter 3

Design and Implementation

3.1 Implementation: CF Compliance and Additional Documentation in Registry

Date last modified: 02/15/13
Contributors: (Doug Jacobsen)

The Registry format needs to be extended to allow additional fields and attributes to be written in the Registry file. This allows developers to write down documentation in a single place, and have parsers propagate those changes through the rest of the required files.

Parsers can include documentation and namelist generation tools. This causes maintenance of documentation and namelists to be a significantly easier task. But requires standardized extensions/modifications for Registry.

Proposed is to migrate Registry into an XML based file, and to modify the Registry parser to handle XML. While XML is not as simple to write as ASCII text, it provides a more structured document that is not driven by the formatting of each line. Although Registry is clean and easy to use, it lacks the extensions required when discussing CF compliance attributes.

XML would provide an easy to extend format for both Registry and auto-documentation. The proposed format is as follows:

```
<?xml version="1.0"?>
<registry>
        <dims>
                <dim name="dimName"
```

```
                units="dimUnits"
                description="Description of Dimension"
                />
        ... more dims ...
    </dims>
    <var_struct name="mesh" time_levs="1">
        <var name="varName"
                type="real"
                units="varUnits"
                dimensions="( dim1 dim2 dim3 )"
                missing_value="-1e34"
                long_name="longVariableName"
                short_name="shortVarName"
                description="Description of Variable"
                />
        ... more vars ...
    </var_struct>
    <nml_record name="record1">
        <nml_option name="config_option1"
                            type="logical"
                            default_value=".true."
                            possible_values=".true. or .false."
                            nml_group="group1"
                            />
        ... more namelist options ...
    </nml_record>
    <streams>
        <stream name="stream1"
                        name_template="stream1.nc"
                        interval="00_01:00:00"
                        frames="1000"
                        type="output" (Input or Restart)
                        description="I/O Stream 1"
                        />
        ... more streams ...
    </streams>
</registry>
```

This format allows all documentation and Registry related information in a single easy to maintain file. The order of the sections (dims, vars, nml_options, streams) does not matter, and neither does the order of the attributes for each specific entry.

This format will be assumed for the remainder of this document.

3.2 Implementation: Easy to use configuration format

Date last modified: 02/15/13
Contributors: (Doug Jacobsen)

Proposed it the use of an additional namelist file. This namelist file would be called something like namelist.input.io. It would remove the burden of I/O configuration from the current namelist.input, and provide the user an easy to configure I/O layer. The proposed format is as follows.

```
&stream_name1
       config_stream_name1_interval = '00_01:00:00'
       config_stream_name1_prefix = 'stream1.nc'
       config_stream_name1_frames_per_file = 1000
       config_stream_name1_type = 'OUTPUT'
/

... more stream groups, one for each stream
... with defaults pulled from Registry.xml

&field_name1
       config_field_name1_stream_name1 = .true. or .false.
       config_field_name1_units = 'varUnits'
       config_field_name1_missing_value = -1e34
       config_field_name1_long_name = 'longVariableName'
       config_field_name1_short_name = 'shrtVarNme'
/
```

In this file, there is a group for each stream, and a group for each variable. Each variable has a config option for each stream that controls if it's included as part of that stream.

This namelist file would not exist in a standard checkout of the trunk, and would get generated by Registry at build time. Upon 'make clean', this namelist file would be removed, as the streams/variables could change between builds.

Additionally, the parser of Registry.xml wouldn't perform any operations on attributes that it doesn't directly use (like description).

At run-time, this new namelist file would be parsed by the I/O layer and handle the addition and removal of fields from streams. It would also handle the addition of field level attributes to the stream output file (if it's

an output stream).

3.3 Implementation: Stream handler

Date last modified: 02/15/13
Contributors: (Doug Jacobsen)

Although we already have the capability of creating I/O streams, an easy to use mpas_stream_handler.F module would be created to allow developers access to the created streams, and their associated alarms.

3.4 Implementation: Auto-documentation parser

Date last modified: 02/15/13
Contributors: (Doug Jacobsen)

Developers will be provided with a parser (probably a python script) of the Registry input file, that will generate a default namelist, and documentation LaTeX files for use in a users guide.

3.5 Potential Implementation: Run-time addition of streams

Date last modified: 02/19/13
Contributors: (Doug Jacobsen)

One possible idea for the run-time addition and removal of streams is to have a general IO namelist group in the I/O namelist file. This group would contain options to handle PIO parameters and general settings that apply to all streams. In addition to this, streams could be defined as follows:

```
&io
       config_io_streams = "input output restart newStream newStream2"
/
```

This config_io_streams string would be broken up, and each substream (space delimited) would represent an independent stream. The namelist parser would then look for a group for each stream, and an option under each field group for each stream.

One potential idea to implement this is to store all streams in a linked list. When adding/creating streams, new streams are appended to this linked list. As fields are created, they are added to the appropriate streams.

By default, all streams are empty. Only fields that are explicitly part of a stream are added to a stream.

Chapter 4

Testing

4.1 Testing and Validation: Run-time I/O Layer

Date last modified: 02/15/13
Contributors: (Doug Jacobsen)