The `yajl_fort` Module

Neil N. Carlson

May 2013
Version 1.0

Abstract

The `yajl_fort` module defines a Fortran interface to the YAJL library. YAJL is an event-driven parser for JSON (JavaScript Object Notation) format data streams written in C. Despite its name, the JSON data format is language independent and makes an excellent data interchange format. It is light weight, flexible, easy for humans to read and write, all of which make it a superior format to XML for many I/O uses.

1 Synopsis

Derived Types

`fyajl_callbacks` (abstract), `fyajl_parser`, `fyajl_status`.

Parameters

`FYAJL_TERMINATE_PARSING`, `FYAJL_CONTINUE_PARSING`, `FYAJL_INTEGER_KIND`, `FYAJL_REAL_KIND`, `FYAJL_ALLOW_COMMENTS`, `FYAJL_DONT_VALIDATE_STRINGS`, `FYAJL_ALLOW_TRAILING_GARBAGE`, `FYAJL_ALLOW_MULTIPLE_DOCUMENTS`, `FYAJL_ALLOW_PARTIAL_DOCUMENT`, `FYAJL_STATUS_OK`, `FYAJL_STATUS_CLIENT_CANCELED`, `FYAJL_STATUS_ERROR`.

Procedures

`fyajl_get_error`, `fyajl_status_to_string`.

2 Prerequisites

The `yajl_fort` module uses YAJL version 2.0. The source code for this library can be obtained from `http://lloyd.github.io/yajl`. The library is also available as a standard system package in all major Linux distributions.

3 Parser Callback Functions

The JSON data language is quite simple. It is built on two basic data structures. An *array* is an ordered list of comma separated *values* enclosed in brackets (`[` and `]`). An *object* is an unordered list of comma separated *name*`:`*value* pairs enclosed in braces (`{` and `}`). A *name* is a string enclosed in double quotes, and a *value* is one of the following: a

string in double quotes, a number (integer or real), a boolean (`true` or `false`), `null`, an *object* or an *array*. Note how the data structures can be nested. Whitespace is insignificant except in strings. At the top level, a valid JSON document consists of a single *object*. See `http://www.json.org` for a detailed description of the syntax.

The C language YAJL parser operates by calling application-defined callback functions in response to the various events encountered while parsing the input stream. The callback functions communicate with each other through a common, application-defined, context data struct, and a void pointer to that data struct is passed to each of the callbacks. In this Fortran interface, this application-defined code/data is implemented by the abstract derived type `fyajl_callbacks`:

```
type, abstract :: fyajl_callbacks
contains
  procedure(cb_no_args), deferred :: start_map
  procedure(cb_no_args), deferred :: end_map
  procedure(cb_string),  deferred :: map_key
  procedure(cb_no_args), deferred :: null_value
  procedure(cb_logical), deferred :: logical_value
  procedure(cb_integer), deferred :: integer_value
  procedure(cb_double),  deferred :: double_value
  procedure(cb_string),  deferred :: string_value
  procedure(cb_no_args), deferred :: start_array
  procedure(cb_no_args), deferred :: end_array
end type fyajl_callbacks
```

Application code extends this type, adding the desired context data components and providing concrete implementations of the callback functions. The required interfaces for the deferred type bound callback functions are:

```
integer function cb_no_args (this)
  class(fyajl_callbacks) :: this
integer function cb_integer (this, value)
  class(fyajl_callbacks) :: this
  integer(FYAJL_INTEGER_KIND), intent(in) :: value
integer function cb_double (this, value)
  class(fyajl_callbacks) :: this
  real(FYAJL_REAL_KIND), intent(in) :: value
integer function cb_logical (this, value)
  class(fyajl_callbacks) :: this
  logical, intent(in) :: value
integer function cb_string (this, value)
  class(fyajl_callbacks) :: this
  character(*,kind=c_char), intent(in) :: value
```

The return value of each function must be either `FYAJL_CONTINUE_PARSING` or `FYAJL_TERMINATE_PARSING`. The latter return value will trigger the parser to terminate with an

error. The kind parameters for integer and real values, `FYAJL_INTEGER_KIND` and `FYAJL_REAL_KIND`, correspond to C's long long and double, and are dictated by the YAJL library. The callbacks are used as follows.

`start_map` is called when a `{` is parsed, marking the start of an *object*.

`end_map` is called when a `}` is parsed, marking the end of an *object*.

`start_array` is called when a `[` is parsed, marking the start of an *array*.

`end_array` is called when a `]` is parsed, marking the end of an *array*.

`map_key` is called when the name of a *name*`:`*value* pair is parsed, and the parsed name string is passed to the function.

`integer_value` is called when an integer *value* is parsed. The value is passed to the function.

`double_value` is called when a real *value* is parsed, and the value is passed to the function.

`string_value` is called when a string *value* is parsed, and the value is passed to the function.

`logical_value` is called when the *value* token `true` or `false` is parsed, and the corresponding Fortran logical value is passed to the function.

`null_value` is called when the *value* token `null` is parsed.

4 Parsing

The derived type `fyajl_parser` and its type bound procedures implement the JSON parser.

```
type, extends(fyajl_callbacks) :: my_callbacks
  ...
end type
type(my_callbacks), target :: callbacks
...
```

As described in the previous section, an application-specific extension of the abstract type must be defined and an instance (here `callbacks`) of that extension initialized if it contains data components.

```
type(fyajl_parser) :: parser
call parser%init (callbacks)
```

Initializes the parser, configuring it to use the specified callbacks. Note that proper finalization of the parser object occurs automatically when the object is deallocated or goes out of scope. Finalization of the callback object is the responsibility of the application.

```
call parser%parse (buffer, stat)
  character(kind=c_char), intent(in) :: buffer(:)
  type(fyajl_status), intent(out) :: stat
```

Parses the chunk of JSON text passed in the `buffer` array. Parsing can be done incrementally via repeated calls to this function, passing successive chunks from the JSON input stream in the `buffer` array. The status of the parser is returned in `stat`; see Error Handling.

```
call parser%parse_complete (stat)
  type(fyajl_status), intent(out) :: stat
```

Parses any remaining internally buffered JSON text. Because the parser is stream-based it needs an explicit end-of-input signal to force it to parse content at the end of the stream that sometimes may exist. The status of parser is returned in `stat`; see Error Handling.

```
parser%bytes_consumed()
```

Returns the number of characters consumed from the buffer in the last call to parse.

4.1 Error Handling

The `parse` and `parse_complete` methods return a `type(fyajl_status)` status value, which equals one of the following module parameters:

FYAJL_STATUS_OK
: No error.

FYAJL_STATUS_ERROR
: A parsing error was encountered; use `fyajl_get_error` to get information about it.

FYAJL_STATUS_CLIENT_CANCELLED
: One of the callback procedures returned `FYAJL_TERMINATE_PARSING`.

The comparison operators `==` and `/=` are defined for `type(fyajl_status)` values.

Several additional functions (not type bound) are provided for error handling.

```
fyajl_get_error(parser, verbose, buffer)
  logical, intent(in) :: verbose
  character(kind=c_char), intent(in) :: buffer(:)
```

Returns a character string describing the the error encountered by the parser. If `verbose` is true, the message will include the portion of the input stream where the error occurred together with an arrow pointing to the specific character. The `buffer` array should contain the chunk of JSON input passed in the last call to `parse`.

```
fyajl_status_to_string(code)
  type(fyajl_status), intent(in) :: code
```

Returns a character string describing the specified status value.

4.2 Parsing Options

The parser supports several options provided by the YAJL library. They are set and unset using the `set_option` and `unset_option` methods after the parser has been initialized:

```
call parser%set_option (option)
call parser%unset_option (option)
```

where `option` is one of the following module parameters. The default for all is unset.

FYAJL_ALLOW_COMMENTS
: JSON does not provide for comments. Setting this option causes the parser to ignore javascript style comments in the input stream. This includes single-line comments that begin with `//` and continue to the end of the line. This is a very useful extention to the JSON standard, but one that is not supported by many JSON parsers.

FYAJL_DONT_VALIDATE_STRINGS
: By default, the parser verifies that all strings are valid UTF-8. This option disables this check, resulting in slightly faster parsing.

FYAJL_ALLOW_TRAILING_GARBAGE
: By default, `parse_complete` verifies that the entire input text has been consumed and will return an error if it finds otherwise. Setting this option will disable this check. This can be useful when parsing a input stream that contains more than one JSON document. In such scenarios, the `bytes_consumed` method is useful for identifying the trailing portion of the input text for subsequent handling.

FYAJL_ALLOW_MULTIPLE_DOCUMENTS
: An instance of a parser normally expects that the input stream consists of a single JSON document. Setting this option changes that behavior and allows an instance to parse an input stream containing multiple documents that are separated by whitespace.

FYAJL_ALLOW_PARTIAL_DOCUMENT
: By default, `parse_complete` verifies that the top level *object* is complete; that is, the closing `}` has been parsed. If it finds otherwise it returns an error. Setting this option disables this check.

5 Bugs

Bug reports and improvement suggestions should be directed to `neil.n.carlson@gmail.com`