Building Structs

May 4, 2019

A spec debugging app should be built using the same publicly available Specs that will be seen in prod. To that end, I'm using Kaitai parsers built from modified copies of the specs.

- The yaml-based `.ksy` is build from public specs
- The parsing is for position/length of fields only, values remain strings (vs casting a Price to decimal and back, etc)

Steps:

1. Get hands on the public specs. I used tabula on the pdf you gave me to make YAMLs representing the "Official" PITCH specs.
2. Convert the specs into Kaitai-compatible `.ksy` output
3. Use the `.ksy` to compile js/py/etc modules that will parse the specs in various environments

[ ]:

[ ]:

[1]:
```
import sys
from pathlib import Path

class CONFIG:
    rootdir = Path.cwd().parent
    specsdir = rootdir / 'structs' / 'specs'
    outfile = rootdir / 'structs' / 'cboe.ksy'
```

[ ]:

0.0.1 Map the spec yaml -> `.ksy` yaml types

This is just mapping the key names and whatnot into Kaitai `.ksy` equivilents.

Before:

```
---
- Name: Symbol Clear
  Section: "4.2"
  Description: >-
      The Symbol Clear message instructs feed recipients to clear all orders for
      the Cboe book in the specified symbol. This message will be sent at
```

```
      startup each day. It would also be distributed in certain recovery events
      such as a data center fail-over.
  Fields:
      - Field Name: Timestamp
        Offset: 0
        Length: 8
        Data Type: Timestamp
        Description: TimeStamp
      - Field Name: Message Type
        Offset: 8
        Length: 1
        Data Type: s
        Description: Symbol Clear Message
      - Field Name: Stock Symbol
        Offset: 9
        Length: 8
        Data Type: Printable ASCII
        Description: Stock symbol right padded with spaces.
```

After:

```
symbol_clear_message:
    doc: >-
        The Symbol Clear message instructs feed recipients to clear all orders for
        the Cboe book in the specified symbol. This message will be sent at
        startup each day. It would also be distributed in certain recovery events
        such as a data center fail-over.
    seq:
      - id: timestamp
        doc: TimeStamp
        type: block('timestamp', 0, 8)
      - id: message_type
        doc: Symbol Clear Message
        type: block('alpha', 8, 1)
      - id: stock_symbol
        doc: Stock symbol right padded with spaces.
        type: block('printable_ascii', 9, 8)
```

[2]:
```
from boltons.strutils import slugify


def convertspec(spec):
    res = {}
    res['doc'] = spec['Description']
    res['seq'] = []
    name = None
    for field in spec['Fields']:
        f = convertfield(field)
```

```
        if f['id'] == 'message_type':
            name = (slugify(field['Description']), field['Data Type'])
        res['seq'].append(f)

    return name, res



def convertfield(field):
    res = {}
    res['id'] = slugify(field['Field Name'])
    res['doc'] = field['Description']
    if res['id'] == 'message_type':
        res['type'] = f"block('alpha', {field['Offset']}, {field['Length']})"
    else:
        dtype = slugify(field['Data Type'])
        res['type'] = f"block('{dtype}', {field['Offset']}, {field['Length']})"
    return res
```

[ ]:

Yaml template that will become our `.ksy` file

[3]:
```
YAML_TEMPLATE = """
meta:
  id: cboe
  # ks-debug: true
  encoding: ASCII
  endian: le
seq:
  - id: records
    type: record(_index, _io.pos)
    # size-eos: true
    eos-error: false
    repeat: eos
instances:
  num_record_entries:
    value: records.size
  record_type_mask:
    value: 0x08
    doc: Magic number to offset the position of the msgtype char in record.
types:
  record:
    params:
      - id: idx
```

```
      type: u4
    - id: ofs
      type: u4
  seq:
    - id: start_of_line
      contents: "S"
      doc: Magic starting char "S"
    - id: raw
      type: strz
      terminator: 0xd
      consume: false
      include: false
    - id: end_of_line
      contents: [0xd, 0xa] # CRLF
  instances:
    type_indicator:
      io: _root._io
      pos: data_offset + _root.record_type_mask
      size: 1
      type: str
    data:
      io: _root._io
      pos: data_offset
      size: data_size
      type:
        switch-on: type_indicator
        cases: {}
    data_offset:
      value: ofs + 1
    data_size:
      value: raw.length
block:
  params:
    - id: type
      type: str
    - id: offset
      type: u2
    - id: length
      type: u2
  seq:
    - id: value
      size: length
      type: strz
"""
```

[ ]:

[ ]:

1 Main Loop

The main thing here is is building the list of message type codes -> their generated type

```
types:
    record:
        ...
        instances: # this stuff was built in previous step
            type_indicator:
                io: _root._io
                pos: data_offset + _root.record_type_mask
                size: 1
                type: str
            data:
                io: _root._io
                pos: data_offset
                size: data_size
                type:
                    switch-on: type_indicator
                    cases:
                        '"B"': trade_break_message    # <-- here
                        '"X"': order_cancel_message

    trade_break_message:
        ...
    order_cancel_message:
        ...
```

[4]:
```
import lib.specs
import lib.yaml


def build_specs(specs, stream):
    ksy = lib.yaml.load(YAML_TEMPLATE)
    cases = ksy['types']['record']['instances']['data']['type']['cases']
    ktypes = ksy['types']

    for spec in specs:
        namecode, newspec = convertspec(spec)
        name, code = namecode
        cases[f'"{code}"'] = name # must have double quotes in string
        ktypes[name]  = newspec

    lib.yaml.dump(ksy, stream)


def build_ksy_specs(base, outpath):
    yaml_specs = list(lib.specs.read_specs_in(base))
```

```
    with Path(outpath).open('w') as stream:
        # should pass an io instance as stream instead
        build_specs(yaml_specs, stream)
```

[ ]:

[5]:
```
build_ksy_specs(CONFIG.specsdir.resolve(), CONFIG.outfile.resolve())
```

[ ]:

Cool, now run `node compile-structs.js` from root dir to generate parsers. It will compile the `/structs/cboe.ksy` into `/structs/compiled/Cboe.js|cboe.py`

[ ]:

[ ]: