Opensource compiler for motorola internal language

Git repo (http)

Project tracker

Orthogonal aspects

Orthogonal aspects must be decoupled to the maximal possible extent:

- output language
- output file structure
- user data structures and field names
- memory allocation

Output files structure (C)

.h

- includes
- enumerations
- struct typedefs
- structures
- prototypes

.c

- includes
- internal declarations
- field packers/unpackers
- message packers/unpackers
- interface packers/unpackers

Pack/unpack signature

```
<pack return type>
pack_<interface name>(
    SignalType signal, /* signal type */
    Param1_t_uunion *p1, /* only present when interface has explicit header with non-autogenerated fields */
    Param2_t_uunion *p2, /* only present when interface has explicit trailer */
    Param3_t_uunion *p3, /* signal data */
    <pdu arg type> pdu, /* pdu output buffer */
    <pdu size arg type> sz /* pdu buffer size */
);
```

```
<unpack return type>
unpack_<interface name>(
    <pdu arg type> pdu, /* input pdu buffer */
    <pdu size arg type> sz, /* pdu buffer size */
    SignalType *signal, /* signal type */
    Param1_t_uunion **p1, /* only present when interface has explicit header
with non-autogenerated fields */
    Param2_t_uunion **p2, /* only present when interface has explicit
trailer */
    Param3_t_uunion **p3, /* signal data */
);
```

When the header is absent Param1 corresponds to the trailer. If there's no trailer, its parameter number corresponds to the signal data.

Interfaces visible from root package, not referenced from other interfaces visible from root package get translated to pack/unpack.

Decoding context

- basic part:
  - bit stream location (byte offset, bit offset)
  - memory allocation context
- custom part:
  - internal variables (managed by 'internal variable assignment' clause)

Field decoder function signature

```
<unpack return type>
unpack_<field name>(
  <decoding context type> * dc, /* data location, updated during unpacking
*/
  <field type> * p /* filled in by this unpack */
);
```

Union member naming

```

Header:

Param1_t_uunion
{
    <interface name>_t <interface name>;
};

Trailer:

Param2_t_uunion
{
    <interface name>_trailer_t <interface name>_trailer;
};

Signal:

Param3_t_uunion
{
    <message name>_t <message name>;
};
```

Message type

- when interface header doesn't have explicit messagetype filed, default 'messagetype : 1 byte' field is added to the tail of the interface header;

Mandatory/optional sections

Mandatory_tagged and mandatory_unordered fields need iei. iei values for mandatory_unordered must be unique in each mandatory_unordered section (?).

- fields in mandatory section go to binary and back as is;
- fields in mandatory_tagged section are prepended by their respective iei when coded. When decoded presence of correct iei is verified;
- fields in mandatory_unordered section are coded like mandatory_tagged. When decoded, they may go in arbitrary order; iei not listed in current mandatory_unordered group terminates decoding of the current group (mt);

All optional fields need iei, which must be unique in each optional section (?).

- fields in optional section have satellite <name>Present field in decoded struct; they're encoded and decoded as mandatory_unordered;
- fields in optional_ordered section imply <name>Present field in decoded struct; they're encoded and decoded as mandatory_tagged;
- fields in optional_repeated form standard list with First, Last, IsAssigned and Length fields; they're encoded in order and decoded in any order;

All sections may interleave, however mt does not diagnose possible ambiguities when first byte of mandatory field may be valid iei. Such diagnostic would be good.

Consecutive mandatory_unordered, optional and optional_repeated sections form a cluster where order of fields is insignificant. Cluster decoding runs for the length of the current scope. Initially this length is determined by PDU length and may be further narrowed down by the message/field own length.

Unknown iei triggers TLV recovery mechanism: mt reads 16 (?) bits after iei and use them as length of unknown iei field.

Only mandatory fields may have LengthRestriction > 1.

Derived interfaces

- if derivation is by inversion, another interface is created and parent's pack-only messages become its unpack-only and vice versa;
- if derivation is by narrowing, another interface is created and extra messages are removed from there;
- derived interface is always represented by the DerivedInterface class;

Validation constraints

Interface

- symbols in pack/unpack sections are visible and refer to messages or interfaces;
- every pack/unpack message has corresponding messagetype;
- header and trailer follow the rules for fields;
- message types are all different;

Field/Message

- types of subfields are visible, parameterized types has required arguments;
- field names are unique in each field contents scope;
- no cyclic dependencies of fields/messages (mt does not allow it);
  - however, even with the current naming rules there may be cyclic dependencies in optional_repeated parts and inside case constructs;
  - generally every optional parts and case content may have cyclic dependencies;

Constants

- expression in constant's definition may only refer to other constants and literals;
- constants may not have recursive definition;

Language features

Definitions

- children: first level of contained objects;
- descendants: all levels of contained objects;

- parent: direct container of the object;
- ancestor: any indirect container of the object;
- sibling: another child of the object's parent;

Namespaces

Namespace is set of object types and the set of rules, that describe how objects of the same name interact in certain scope.

- packages (P): last definition is effective in the package scope;
- constants, fields, messages, interfaces (CFMI): must be unique in visible scope;
- subfields: subfields must be unique in the field contents scope;

Scopes

- package: children of the package object;
- natural scope: package + package scopes of ancestors;
- visible scope: package + visible scopes of used packages + visible scopes of ancestors;
- field/message/interface header/interface pack-unpack/interface trailer;
- field contents: each level of nested braces in field/message/interface (header and trailer parts) definition;

Packages

- package nesting is unrestricted;
- use does not affect P namespace;
  - if A contains B, B contains C and A use B, one cannot write A use C;
- use adds visible scope of the used package to the current visible scope in CFMI namespace;
  - use directive in transitive, i.e. A use B and B use C means A use C;
- only package from natural scope may be used, i.e. package may use only children of its ancestors and its children;

- original parser has issues with packages:
  - it treats root package differently than named packages;
  - sibling packages don't merge and don't collide, one defined later is effective, others are ignored;

Naming for C output

There are two generic rules:

- type names are generated with suffix '_t' added to the base name;
- when there's only one subfield in the field it gets reduced, like there's no container structure. Subfield gets basic name of its container;
  - 'one subfield' is literally one subfield. Presence of the special-purpose subfields that would not get into the structure or constant fillers cancels reduction.

Field/message type

<C field type> = <Field name>_t

Ex:

```
field A
{
...
}

typedef struct {...} A_t;
```

Enum

Enum type:

- if this is the only subfield (so that reduction takes place), <C enum type> = <C field type>;
- otherwise <C enum type> = <Field name>_<subfield name>_t;

<C Enum value> = <C enum type>_<Enum value>

Ex:

```
field A
{
    a : 1 byte { V1 = 1, V2 = 2 };
}
field B
{
    a : 1 byte { V1 = 1, V2 = 2 },
    b : 1 byte { V1 = 1, V2 = 2 };
}

typedef enum { A_t_V1 = 1, A_t_V2 = 2 } A_t; // reduction
typedef enum { B_a_t_V1 = 1, B_a_t_V2 = 2 } B_a_t;
```

Case

- <C choice type> = <C container type>_<tag subfield name>_choice{index, if > 1}
- <C choice type_t> = <C choice type>_t
- case labels (only literals, tag field enum values or global constants may be used):
  - for numeric labels <C case name> = case<case label>
  - for enum labels <C case name> = <enum label>

Choice is always represented by the following structure:

```
struct <C choice type_t> { <C choice type_t>_tag_enum _tag; <C choice
type_t>_uunion U; };
typedef enum { <C choice type_t>_tag_enum_<C case name>, ... } <C choice
type_t>_tag_enum;
union <C choice type_t>_uunion { <C choice type>_<C case name>_t <C case
name>; ... };
```

Ex: see case.pdu

Schedule

- finalize naming for the following items:
    - exported constants;
    - cases;
    - ranges;
    - arrays of primitive types;
    - strings;
    - interfaces, derived and nested;
    - ?
- separate compiler frontend and backend;
- implement validators;
- implement pack/unpack;
    - interfaces;
    - optional/mandatory;
    - alignment;
    - type instantiation and deep referencing;
- test;
- presentation;