

# Opensource compiler for motorola internal language

[Git repo \(http\)](#)

## 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.

## 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_union
{
    <interface name>_trailer_t <interface name>_trailer;
};

Signal:

Param3_t_union
{
    <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 m\_u group terminates decoding of the current group (mt implementation);

All mandatory 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.

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;

Consecutive optional and optional\_repeated sections form a cluster where order of fields is insignificant. Cluster decoding is interrupted by the unlisted iei (?).

## 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;

## 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;

## 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;

From:

<http://wiki.osll.ru/> - **Open Source & Linux Lab**

Permanent link:

<http://wiki.osll.ru/doku.php/etc:users:jcmvbkbc:isl?rev=1281640553>

Last update: **2010/08/12 23:15**

