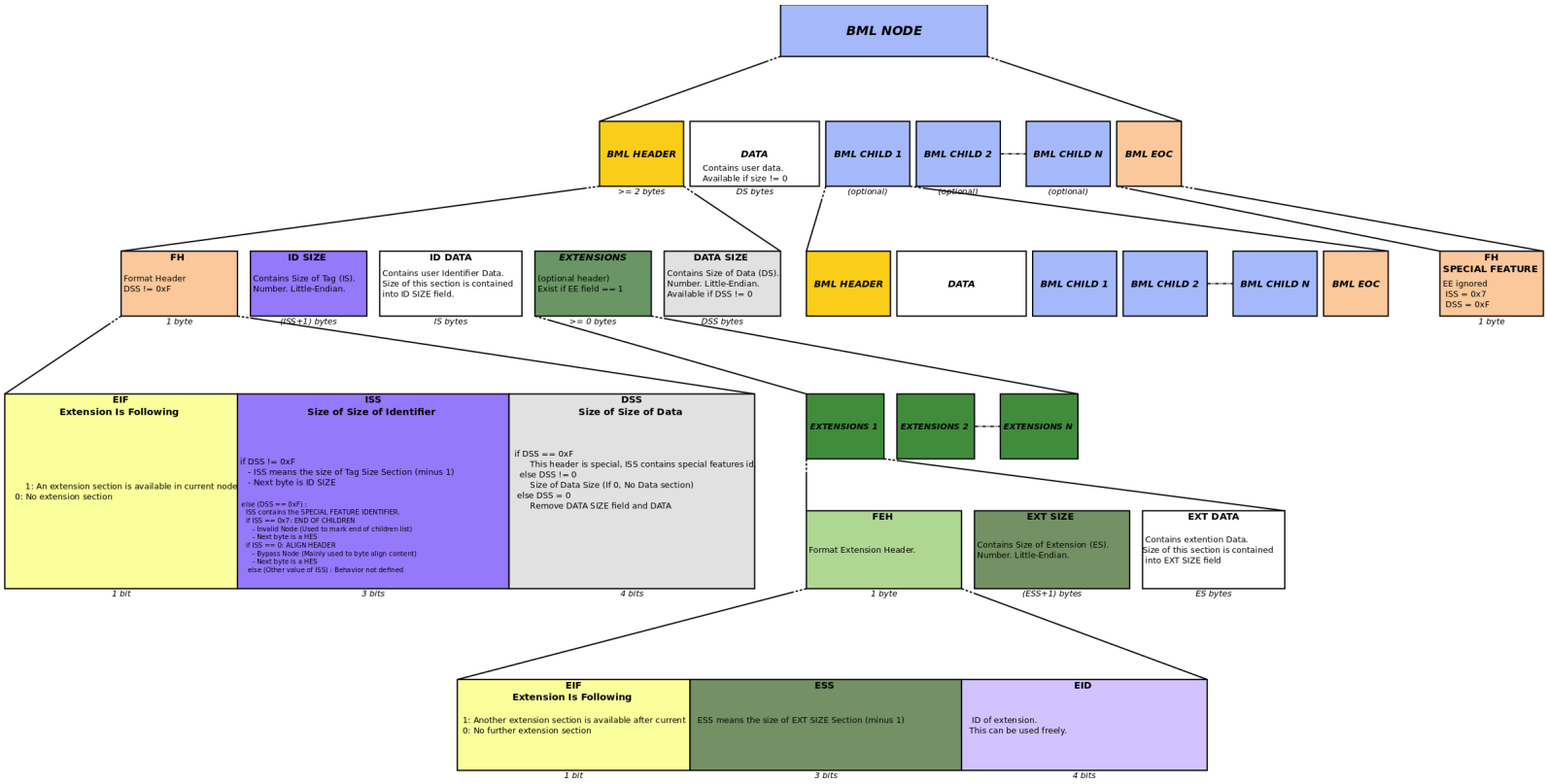
What is BML?

- **BML** stands for Binary Markup Language
- **BML** was designed to carry data.
- **BML** was designed to be used in a file or in digital communication (socket, digital bus, ...).
- **BML** was designed to be light for encoding and decoding.
- **BML** tags are not predefined. You must define your own tags. Tags can be string, integer, or custom binary data.
- **BML** elements handle child elements as XML.

What are the differences between BML and XML ?

- An XML document is a string of character. A **BML** document is binary.
- An XML document is divided into Markup and Content. A **BML** document is divided into Headers with size information and Content
- **BML** does not handle attributes.
- **BML** handles Extensions. Extensions can be used to add custom fields to content or headers.
- **BML** does not have any illegal character (unlike XML: &, < and >)

BML - a binary XML format 1.1



Section Element BML

Availability	**Sections**	**Size** (bytes)	**Description**
x1	BML Header	> 2	Contains Tag, Extensions and Data description
x1 (optional)	Data	DS	Contains Data Content
xN (optional)	Childs	>= 0	Contains children of current node using Node BML format
x1	EOC	1 byte	Ends list of children

- **Section Header BML**

Availability	Sections	Size (bytes)	Description
x1	FH	1	Describe availability and size of fields: Extensions, Id Size and Data Size
x1	Id Size	ISS + 1	Contains Size of Id (IS). Little-Endian.
x1	Id Data	IS	Contains Tag
x N Optional	Extensions	>= 0	Contains auxiliary data
x 1 Optional	Data Size	DSS	Contains Size of Data (DS). Little-Endian.

  - Section FH

	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
FH	**EIF**	**ISS2**	**ISS1**	**ISS0**	**DSS3**	**DSS2**	**DSS1**	**DSS0**

- **EIF**
  - Format: 1 bit.
  - 1: An extension section is available in current node
  - 0: No extension section
- **ISS**
  - Format: 3 bits unsigned integer.
  - if **DSS** != 0xF :
  -
    - ISS means the size of Tag Size Section (minus 1)
  - else :
    - ISS == 0x7: Invalid Node (Used to mark end of children list)
    - ISS == 0: Bypass Node (Mainly used to byte align content)
    - Other behavior not defined

- **DSS**
  - Format: 4 bits unsigned integer.
  - if **DSS** == 0xF
    - This node is special, ISS contains special features

- else
  - Size of Data Size (If 0, No Data section)

- Section Header Extension

Availability	Sections	Size (bytes)	Description
x1	FEH	1	Describe current extension
x1	Ext Size	ESS + 1	Contains Size of EXT (ES). Little-Endian.
x1	Ext Data	ES	Contains Data of EXT

- Section FEH

	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
FEH	**EIF**	**ESS2**	**ESS1**	**ESS0**	**EID3**	**EID2**	**EID1**	**EID0**

- **EIF**
  - Format: 1 bit.
  - 1: Another extension section is available after current
  - 0: No further extension section
- **ESS**
  - Format: 3 bits unsigned integer.
  - ESS means the size of EXT Size Section (minus 1)
- **EID**
  - Format: 4 bits unsigned integer.
  - ID of extension. This can be used freely. (In current decoding library used to mix order of first 16 extensions)

- File Header BML

Availability	**Sections**	**Size** (bytes)	**Description**
x1	Preamble BML	4	Contains 'B' 'M' 'L' and '\0' (version of format)
xN	Node BML	>3	Node BML

BML - Libraries

Two libraries can be used to read / write and use the BML data model:

- **BML_NODE** (BML Usage Class written in C++). Similar to XML DOM
- **SAB** (Simple API for BML): Language C, similar to XML SAX

The **SAB** and **BML_NODE** methods each adopt a very different strategy for analyzing the syntax of BML documents, so they are used in different contexts.

BML_NODE loads an entire BML document into a data structure, which can then be manipulated and converted back to BML. However, for this purpose, the size of the structure representing the BML document must not be greater than (or not too much) what RAM can contain. The **SAB** method then provides an alternative in cases where BML documents are very large.

Both **BML_NODE** and **SAB** are parts of libbml.

Download: https://github.com/seagnal/bml
Homepage: http://www.gnu-log.net/home/bml

SEAGNAL is involved into libbml development. Libbml is licensed under LGPLv3.

La librairie C++ : **BML_NODE**

The whole BML structure is contained into memory and you can use methods of node<T, G> class in order to manipulate the BML node.

BML_NODE allows you :

- To manipulate BML node
  - Update data from external buffer.
  - Copy data to external buffer.
  - Create and delete child
  - Access child
  - Find childs
  - Create and delete extensions
- to write BML document or buffer through node_writer<G> class.
- to read BML document or buffer through node_parser<G> class.

Creating node

```
#include <bml_node.hh>

/* Create an empty node with no id */
node<uint32_t, std::shared_ptr> c_example_node;

/* Create an empty node with ID = 1234 */
{
     node<uint32_t, std::shared_ptr> c_node(1234);
}

/* Create a node with data */
{
     uint32_t i_data = 10;
     node<uint32_t, std::shared_ptr> c_node(20, i_data);
}

/* Set id of current node */
c_example_node.set_id(1234);
```

SEAGNAL®

Accessing childs

```
typedef std::shared_ptr< node<uint32_t, std::shared_ptr> > smart_pointer_to_node_t;
  typedef node<uint32_t, std::shared_ptr> node_t;

  /* Accessing child with ID 6 and index 0. Node is automatically created if it does not
exist */
  node_t & ref_to_child  = c_example_node(6);
  node_t & ref_to_child2 = c_example_node(6,0);
  smart_pointer_to_node_t smart_pointer_to_child = c_example_node.get(6,0,true);

  /* Accessing child with ID 6 and index 10. DO NOT WORK if index 9,8,..,1,0 is not
available. */
  try {
        node_t & ref_to_child3 = c_example_node(6,10);
        smart_pointer_to_node_t smart_pointer_to_child2 = c_example_node.get(6,10,true);
  } catch(...) {
  }
  /* Accessing child with ID 6 and index 0. Node is not automatically created if it does not
exist */
  smart_pointer_to_node_t smart_pointer_to_child3 = c_example_node.get(6,0, false);

  /* Parse child with specific id */
  {
       node<uint32_t, std::shared_ptr>::find_result c_result;
       node<uint32_t, std::shared_ptr>::find_iterator c_it;
       c_result = c_example_node.find(6);

       for(c_it = c_result.begin(); c_it != c_result.end(); c_it++) {
          int tmp = (*c_it)->second->get_data<int>();
       }
  }

 /* Parse all childs */
  {
       node<uint32_t, std::shared_ptr>::find_result c_result;
       node<uint32_t, std::shared_ptr>::find_iterator c_it;
       c_result = c_example_node.childs();

       for(c_it = c_result.begin(); c_it != c_result.end(); c_it++) {
          int tmp = (*c_it)->second->get_data<int>();
       }
  }

  /* Update data of child */
  smart_pointer_to_child->set_data(i_data8);
```

What is BML?

SEAGNAL®

Getting and setting data

```
/* Set data of current node (uint64_t)*/
 c_example_node.set_data(i_udata64);

 /* Get data of current node (uint64_t)*/
 uint64_t i_udata64_2 = c_example_node.get_data<uint64_t>();
 //std::cout <<i_udata64 << " "<<i_udata64_2<< std::endl;
 ck_assert(i_udata64_2 == i_udata64);

 /* Set data on it with cast (int32_t)*/
 c_example_node.set_data<uint32_t>(i_data8);

 /* Get data of current node (int32_t)*/
 uint32_t i_tmp = c_example_node.get_data<uint32_t>();

 /* resize data of node */
 c_example_node.resize(4);

 /* Mapping data for update */
 {
      uint32_t * pi_data = c_example_node.mmap<uint32_t>();
      *pi_data = 10;
 }

 /* Accessing data through a type reference */
 uint32_t & i_data = c_example_node.map<uint32_t>();
 i_data++;
 ck_assert(i_data == 11);


 /* Using memcpy property */
 {
      char ai_data[] = "datadata";
      /* Copy data from buffer to node */
      c_example_node.memcpy_from_buffer(ai_data, sizeof(ai_data));

      /* Copy data from node to buffer */
      c_example_node.memcpy_to_buffer(ai_data, sizeof(ai_data));
 }
```

Segment & Resources

Segment and resources handle data block. Those classes are used as a data block abstraction layer. All data access are performed through those two classes.

```
  /* Create node from segment */
  {
        std::shared_ptr<node_resource<std::shared_ptr> > c_resource(new
node_alloc_resource<std::shared_ptr>(1024));
        node_resource_segment<std::shared_ptr> c_segment(c_resource);
        node<uint32_t, std::shared_ptr> c_example_node2(1234, c_segment, 10, 20);
  }

  /* Accessing data through mmap of segment */
  {
        node_resource_segment<std::shared_ptr> segment = c_example_node.get_segment();
        char * pi_data = segment.mmap();
        *pi_data = 10;
  }

  /* Accessing data through memcpy of buffer */
  {
        char ai_data[] = "datadata";
        node_resource_segment<std::shared_ptr> c_segment_node = c_example_node.get_segment();
        c_segment_node.memcpy_from_buffer(ai_data, sizeof(ai_data));
        c_segment_node.memcpy_to_buffer(ai_data, sizeof(ai_data));
  }

  /* Accessing data through memcpy of segment */
  {
        std::shared_ptr<node_resource<std::shared_ptr> > c_resource(new
node_alloc_resource<std::shared_ptr>(1024));
        node_resource_segment<std::shared_ptr> c_segment(c_resource);

        node_resource_segment<std::shared_ptr> c_segment_node = c_example_node.get_segment();
        c_segment_node.resize(c_segment.size());
        c_segment_node.memcpy_from_segment(c_segment);
        c_segment_node.memcpy_to_segment(c_segment);
  }
```

SEAGNAL®

Writing node to buffer

```
{
      node<uint32_t, std::shared_ptr> c_node(1234);
      c_node.set_data<uint32_t>(10);
      c_node(1).set_data<uint32_t>(1);
      c_node(2).set_data<uint32_t>(2);
      c_node(3).set_data<uint32_t>(3);

      char ac_buffer[1024];
      node_buffer_writer<std::shared_ptr> c_writer(ac_buffer, sizeof(ac_buffer));
      c_node.to_writer(c_writer);
}
```

Loading node from buffer

```
{
      node<uint32_t, std::shared_ptr> c_node_read;
      node_buffer_parser<std::shared_ptr> c_parser(ac_buffer, sizeof(ac_buffer));
      c_node_read.from_parser(c_parser);
      c_node_read.dump();
      ck_assert(c_node.get_data<uint32_t>() == 10);
}
```

SEAGNAL®

Writing node to file

```
{
      node<uint32_t, std::shared_ptr> c_node(1234);
      c_node.set_data<uint32_t>(10);
      c_node(1).set_data<uint32_t>(1);
      c_node(2).set_data<uint32_t>(2);
      c_node(3).set_data<uint32_t>(3);

      node_file_writer<std::shared_ptr> c_writer("/tmp/data.bml");
      c_node.to_writer(c_writer);
}
```

Loading node from file

```
{
      node<uint32_t, std::shared_ptr> c_node;
      node_file_parser<std::shared_ptr> c_parser("/tmp/data.bml");
      c_node.from_parser(c_parser);
      ck_assert(c_node.get_data<uint32_t>() == 10);
}
```

Set and get extension

```
{
      node<uint32_t, std::shared_ptr> c_node(1234);
      uint64_t i_data_ext = 123456;
      c_node.set_ext<uint64_t>(1, i_data_ext);

      uint64_t i_data_ext_get;
      i_data_ext_get = c_node.get_ext<uint64_t>(1);

      ck_assert(i_data_ext_get == i_data_ext);
}
```

Find a child

```
{
      node<uint32_t, std::shared_ptr> c_node(1234);
      c_node.set_data<uint32_t>(10);
      c_node(1).set_data<uint32_t>(1);
      c_node(2,0).set_data<uint32_t>(4321);
      c_node(2,1).set_data<uint32_t>(4321);

      node<uint32_t, std::shared_ptr>::find_result c_result;
      node<uint32_t, std::shared_ptr>::find_iterator c_it;
      c_result = c_node.find(2);

      for(c_it = c_result.begin(); c_it != c_result.end(); c_it++) {
      ck_assert( (*c_it)->second->get_data<uint32_t>() == 4321);
      }
      ck_assert(c_result.size() == 2);
}
```

La librairie C : **SAB**

BML_PARSER

Solely to read a XML document. The SAB parser runs through the document and execute callback methods of the user. There are methods for start/end of a document, element and so on. The streaming nature of the SAB-parser object makes it a better choice for reading large BML documents.

Usage:
First a BML parser callback structure must be filled.

```
struct bml_parser_callbacks {
    /*! Start BML node callback. Means new child of current */
    bml_cb_start_element cb_start_element;
    /*! End BML node callback. Means that we reach end of childs of current node. Moving to
parent. */
    bml_cb_end_element cb_end_element;
    /*! Read data callback. */
    bml_cb_io_read cb_io_read;
    /*! Seek inside data callback.
      Can be NULL if cb_io_read is cb_new_data is not null.
        Parser will jump of bml content, extracting offset for further direct access.  */
    bml_cb_io_seek cb_io_seek;
    /*! Memory allocation of Id callback.
      Callback callback prior to Id read (through bml_cb_io_read) in order to let user control
where data will be copied.
        Reducing number number of memcopy */
    bml_cb_new_id cb_new_id;
    /*! Memory allocation of data callback
      Callback callback prior to data read (through bml_cb_io_read) in order to let user
control where data will be copied.
        Reducing number number of memcopy.
        Can be NULL, if user do not want to copy data during BML parse. */
    bml_cb_new_data cb_new_data;
    /*! Memory allocation of extension data callback
      Callback callback prior to an extension read (through bml_cb_io_read) in order to let
user control where data will be copied.
        Reducing number number of memcopy */
    bml_cb_new_ext cb_new_ext;
};
```

Then three methods can be called:

- To initialize the parser (providing user callbacks to BML_PARSER)

```
int bml_parser_init(struct bml_parser * in_ps_parser,
        struct bml_parser_callbacks * in_ps_cb, void * in_pv_user_arg);
```

- To parse the document. This function will block until a whole root is parsed

```
int bml_parser_run(struct bml_parser * in_ps_parser);
```

- To stop the document parsing. This function can only be executed from callbacks.

```
int bml_parser_stop(struct bml_parser * in_ps_parser);
```

BML_WRITER

Writing BML with a SAB-writer object is an alternative to outputting a BML object. The document tree needs to be completely built and stored in memory before you can output it. In contrast, the SAB writer object only needs enough memory to handle the largest single element in your BML output. The streaming nature of the SAB-writer object makes it a better choice for writing large BML documents.

Usage:
First a BML writer callback structure must be filled

```
struct bml_writer_callbacks {
    /*! Write data callback */
    bml_cb_io_write cb_io_write;
};
```

Then five methods can be called:

- Initialization

```
int bml_writer_init(struct bml_writer * in_ps_writer,
        struct bml_writer_callbacks * in_ps_cb, void * in_pv_user_arg);
```

- Write of a BML node Header & Data without its childs

```
int bml_writer_element_start(struct bml_writer * in_ps_writer, void * in_pv_id,
        bml_size_t in_sz_id, void * in_pv_data, bml_size_t in_sz_data,
        struct bml_ext * in_as_ext, uint8_t in_i_nb_ext,
        bml_size_t in_i_length_id_size, bml_size_t in_i_length_siz_size, bml_size_t
in_i_length_ext_size_size);
```

- Write BML End of Child

```
int bml_writer_element_end(struct bml_writer * in_ps_writer);
```

- Data alignment

```
int bml_writer_align(struct bml_writer * in_ps_writer,
        bml_size_t in_i_alignment);
int bml_writer_prealign(struct bml_writer * in_ps_writer, bml_size_t in_i_alignment,
bml_size_t in_i_offset);
```

Example of SAB parser and writer can be found into bml.cc of python bindings.

Download libbml from source code

```
$ git clone https://github.com/seagnal/bml
```

Installing libbml

```
$ cd libbml
$ bash ./autogen.sh
$ ./configure
$ sudo make install
```

BML - Bindings

In order to use the functionalities proposed by the BML and its libraries, in languages other than C / C ++, bindings of these libraries are available:

- In Python Language (DOM with SAB library)
- In language Octave / Matlab (SAB)

Using BML with Python 3.x

Using bindings from source code

Execute:

```
$ sudo python3 ./setup.py install
```

Usage - Create a node

```
from bml_tools import bml
a = bml.node(10)
dat =struct.pack('BBB', 10, 11, 35);
a.data = dat;
```

Get id

```
a.id
```

Get data

```
a.data
```

Accessing child

```
a.get(11)
```

Accessing child data

```
a.get(11).data
```

Usage - Append childs

```
b = bml.node(12)
b.data = "trop cool 12";
a.append(b);
b = bml.node(11)
b.data = "trop cool 11";
a.append(b);
```

Use file to read or write BML data

```
# Write to file
wr = bml_file('/tmp/test.bml','w');
a.to_writer(wr,0);
wr.close()

# Read from file
rd = bml_file('/tmp/test.bml','r');
ar = bml.node();
ar.from_parser(rd);
```

Use socket to read or write BML data

```
# Opening Unix socket (Other sockets can be used, ex:TCP, UDP, ...)
s1, s2 = socket.socketpair()
client1 = bml_socket(s1);
client2 = bml_socket(s2);

# Write BML node to a socket
try:
        a.to_writer(client1,0);
except bml.exception as e:
        print(e.message)
        sys.exit(-1)

# Get node on other side of socket
ar = bml.node();
ar.from_parser(client2);
```

Custom writer container

```
class wr_test(bml.writer):
        def io_write(self, in_data):
        print("write " +str(len(in_data)) + " bytes :" + str(in_data.tobytes()))
        return 0
wr = wr_test();
a.to_writer(wr,0);
```

Using BML with Octave

Using bindings from source code

1. Execute compile.m from octave or matlab inside octave folder
2. Use addpath() method to add path to octave bml folder

Writing BML into a file

```
filename=[getenv("HOME") "/bml_test.dat"];

 printf('TEST WRITE: %s\n',filename);

 % Opening file in order to write data into
 fid = bml_open(filename, 'w');

 % Inserting node inside file
 bml_write(fid, struct('id', 1,  'data', 'data1'));
 bml_write(fid, struct('id', 10, 'data', "data10"));
 bml_write(fid, struct('id', 1,  'data',  uint64(20000)));
 bml_write(fid, struct('id', 10, 'data', "data10.2"));
 bml_write(fid, struct('id', 11, 'data', []));
 bml_write(fid, struct('id', 12,    'data',  uint32([5,6])));

 % Inserting node inside file with extension
 node = struct('id', 1, 'data', uint64(30000));
 node.ext = {[],[],uint64([3])};
 bml_write(fid, node);

 % Inserting node inside file with 2 childs
 a = struct('id', 13, 'data', "data child 1");
 b = struct('id', 14, 'data', "data child 2");
 bml_write(fid, struct('id', 10, 'data', "data parent" , "childs", [a, b]));

 % Closing file
 bml_close(fid);
```

Reading BML from a file

```
% Opening file in order to read data from
 fid = bml_open(filename, 'r');
 cnt = 1;
 count = 0;
 ii = 1;
 id = zeros(1500,1);

 % Loop while there is node to read
 while cnt
  % Read nodes from files. Return multiple nodes
  [res , cnt] = bml_read(fid);
  % Node user read function
  count = count + display_node(res,cnt);
 end
 bml_close(fid);
```

```
% octave
% Display a BML node (Id, Data and childs).
function count = display_node(res, cnt, level)

    if !exist('level')
   level = 0;
    end
    count = 0;
    for ii = 1:cnt
        r = res(ii);
       % Cast Id from char array
        if length(r.id) == 4
               id_str =dec2hex(double(typecast(r.id,'uint32')));
        elseif length(r.id) == 8
               id_str =dec2hex(double(typecast(r.id,'uint64')));
        else
               id_str = char(r.id);
        end

       % Print level of child
        for tt = 1:level
               printf("-");
        end

        % Print datadata
        if length(r.data)
               printf("id:%s off:%d(%d) data:%s(%d) nb_childs:%d :%s\n",id_str, r.offset,
r.size, dec2hex(double(char(r.data)))',length(r.data), size(r.childs,2),  char(r.data) );
        else
               printf("id:%s off:%d(%d) nb_ext:%d nb_childs:%d\n",id_str, r.offset, r.size,
length(r.ext), size(r.childs,2) );
        end
```

```
        % recursive display call in order to show childs
        count = count + display_node(r.childs, size(r.childs,2), level+1);
        count++;
    end
```