lime::md::**market_feed_interface**

#include <library/citrius.h>

```
template
<
    network_mode T
> class market_feed_interface;
```

Creates a network interface which can be used to join Citrius 2.0 multicasts using either kernel based networking or kernel bypass (efvi) based networking.

Template parameter:

T – A value from the enumeration `network_mode`. Specifies kernel or kernel bypass (efvi).

```
enum class network_mode : std::uint32_t
{
    undefined     = 0,
    kernel_bypass = 1,
    kernel        = 2
};
```

Specializations:

```
template class lime::md::market_feed_interface<lime::md::network_mode::kernel>;
template class lime::md::market_feed_interface<lime::md::network_mode::kernel_bypass>;
```

Public Member Functions:

`market_feed_interface()`	Constructs the market_feed_interface using default configuration
`market_feed_interface(configuration const &);`	Constructs the market_feed_interface using custom configuration
`~market_feed_interface();`	Destructs the market_feed_interface
`void poll();`	Polls the sockets established through this market_feed_interface
`void receive();`	Processes the messages contained within the next multicast packet available to any socket established through this market_feed_interface
`bool is_valid() const;`	Returns true if this instance of market_feed_interface is valid

`template <market_feed_concept recipient, typename ... Ts>`
`std::unique_ptr<recipient> join_multicast(std::string, Ts && ...);`
	Creates an instance of the specified recipient type providing Ts … to the constructor of that recipient. Joins the specified multicast and routes all messages received on that multicast to the newly created instance of the recipient.

lime::md::market_feed_interface<T>::**market_feed_interface**

market_feed_interface() : market_feed_interface(configuration()){}	(1)
market_feed_interface(configuration const &);	(2)

Constructs a new market_feed_interface.

1) The default constructor delegates to (2) using the default configuration settings.
2) constructs using the specified configuration settings.

struct lime::md::market_feed_interface<T>::**configuration**

```
static auto constexpr default_max_socket_capacity      = 256ull;
static auto constexpr default_max_buffer_heap_capacity = 1ull << 16;

struct configuration
{
    std::string   networkInterfaceName_;
    std::uint64_t maxSocketCapacity_{default_max_socket_capacity};
    std::uint64_t maxBufferHeapCapacity_{default_max_buffer_heap_capacity};
};
```

std::string configuration::networkInterfaceName_;	The name of the physical network interface to use. Example: “eth0” If this value is empty then a best attempt is used to select an appropriate existing physical network interface name.
std::uint64_t configuration::maxSocketCapacity_;	The maximum number of multicast which can be join using this instance of a market_feed_interface<T>.
std::uint64_t configuration::maxBufferHeapCapacity_;	Each instance of market_feed_interface<T> has its own dedicated network packet buffer heap. This value determines how large this heap shall be.

lime::md::market_feed_interface<T>::**~market_feed_interface**

`~market_feed_interface();`	(1)

1) A destructor. Destructs the market_feed_interface and leaves any joined multicasts.

`lime::md::market_feed_interface<T>::`**`poll`**

`void poll();`	(1)

1) Each instance of a `market_feed_interface<T>` has one poller. For `T = network_mode::kernel`, the poller uses `::epoll()`. For `T = network_mode::kernel_bypass`, the poller uses `ef_eventq_poll()`.

Invoking `market_feed_interface<T>::poll()` will poll the sockets associated with this instance and schedule the selected sockets to receive those packets. To receive those packets see `market_feed_interface<T>::receive()`.

lime::md::market_feed_interface<T>::**receive**

`void receive();`	(1)

1) Processes the next packet available to the next socket which has been selected via a call to `lime::md::market_feed_interface<T>::poll()`. The packet is parsed and each message is forwarded to the message receiver associated with the specific socket.

Note: `market_feed_interface<T>::receive()` is thread safe. Therefore, calling `market_feed_interface<T>::receive()` from multiple threads in parallel can result in parallel message callbacks from two or more associated `citrius_market_feed<T>`.

lime::md::market_feed_interface<T>::**is_valid**

`bool is_valid() const;`	(1)

1) Returns `true` if the `market_feed_interface<T>` is in a valid state. Returns `false`, otherwise.

lime::md::market_feed_interface<T>::**join_multicast**

```
template <market_feed_concept recipient, typename ... Ts>               (1)
std::unique_ptr<recipient> join_multicast(std::string, Ts && …);
```

Creates an instance of type `recipient` (forwarding `Ts && …` as arguments to the recipient constructor) and returns a `std::unique_ptr` to that instance. Joins the specified multicast and routes all received Citrius 2.0 messages from that multicast to that instance of `recipient`. Upon destruction of the instance of `recipient`, the associated socket will be closed and the multicast join will be terminated.

The `recipient` type must satisfy `market_feed_concept`.

lime::md::citrius_market_feed

#include <library/citrius.h>

```
template
<
    typename recipient,
    market_feed_traits_concept market_feed_traits_type,
    bool allow_polymorphic_recipient = false
> class market_feed;


template <typename recipient>
using citrius_market_feed = market_feed<recipient, lime::md::citrius_market_feed_traits>;
```

Creates a new citrius_market_feed instance and routes all Citrius 2.0 messages received on from it to an instance of the specified recipient type. citrius_market_feed is a partial specialization of class market_feed<>.

Template parameter:

recipient – A type which can receive Citrius 2.0 messages by implementing an accessible overload of recipient::operator()(message_type const &) const;

citrius_market_feed(configuration const &, event_handlers const &);	Constructs the object and configures and assigns event handlers as requested.
void close();	Closes the underlying socket, leaving the multicast

Concepts:

```
template <typename T> concept market_feed_concept =
std::is_base_of_v<market_feed<typename T::recipient, typename T::market_feed_traits,
T::allow_polymorphic_target>, T>;
```

Notes:

citrius_market_feed routes Citrius messages to the receiver using a reinterpret_cast<>. Therefore the recipient type must not be polymorphic as reinterpret_cast<> would fail. If recipient type must be polymorphic then

struct lime::md::citrius_market_feed<T>::**configuration**

```
static auto constexpr default_receive_buffer_size = ((1ull << 20) * 64);

struct configuration
{
    std::string   socketAddress_;
    std::uint64_t receiveBufferSize_{default_receive_buffer_size};
};
```

std::string configuration::socketAddress_;	The multicast address to join.
std::uint64_t configuration::receiveBufferSize_;	Sets the underlying UDP socket’s RX buffer size.

struct lime::md::citrius_market_feed<T>::**event_handlers**

```
using data_error_handler = std::function<void(market_feed const &, std::span<char const>)>;
using close_handler = std::function<void(market_feed const &)>;
using sequence_gap_handler = std::function<void(market_feed const &, sequence_number, std::uint64_t)>;
```

```
struct event_handlers
{
    data_error_handler   dataErrorHandler_;
    sequence_gap_handler sequenceGapHandler_;
    close_handler        closeHandler_;
};
```

data_error_handler dataErrorHandler_;	Callback when invalid Citrius 2.0 data has been encountered.
sequence_gap_handler sequenceGapHandler_;	Callback when a sequence gap (packet loss) has happened on the associated Citrius 2.0 multicast feed.
close_handler closeHandler_;	Callback when the citrius_market_feed closes.