Embedded Software

Inter-thread communication (Intra-process communication)

AARHUS UNIVERSITY

Agenda

- Communication design challenges
- Message Queue and Handler Design
- Consequences

Communication design challenges

- Individual threads wait for a condition to become true
- Enter and leave critical sections using mutexes or semaphores
  - May happen multiple times in the space of one thread loop iteration
- May even hold multiple resources which have to be synchronized between threads
  - The sequence in which resources are taken must be thought through.
- ***Consequence***
  - A design challenge ensuring that no deadlocks or timing issues exist
  - Readability easily becomes an issue too
  - High code complexity is the outcome

What we need...

- We want an approach where
  - *all* processing within a thread must *not* require locking
  - however *other* threads must be able to pass control and/or data to a specific thread via some mechanism.
  - *multiple* threads may concurrently decide to pass such control and/or data

A step backwards

- What is it in fact we are doing and what?
  - *Perform some action when a given condition becomes true or we get signaled*

We want events (messages)!

Event Driven Programming

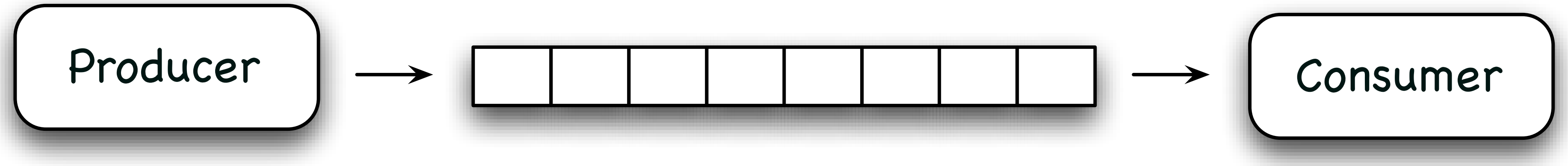
- Reactionary programming
    - ***Each incoming message is processed by a specific handler***
        - E.g. its a handler - *it reacts someone must take initiative!*
    - Types
        - Censor input
            - Temperature exceeded message → Turn down heat
            - Car detected wanting to enter car park message → Open garage door
        - Signal input
            - Exit button in GUI message → Exit program

Event Driven Programming (Event = Message)

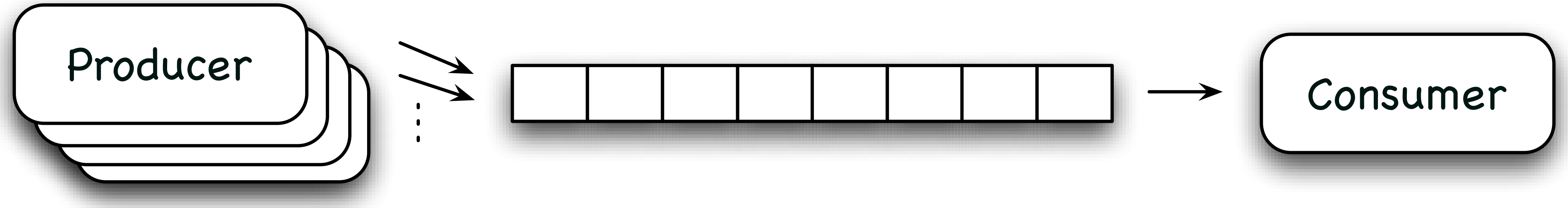
- Can be viewed as a two phase process
  - Acquire/Select new message
    - Handled by a *Message Queue* and ensures that a number messages can be in “queue” at a time
  - Process new message in handler
    - Handled by casing out on the specific message

Message Queue & Handler design

Resembles the "Producer & Consumer problem"



- The producer-consumer problem
  - A producer thread produces buffer items
  - A consumer thread consumes them
- Applied to our problem we get



Further requirements for *our* Message Queue

- If the receiving queue is full, then the thread or threads wishing to pass control and/or data must block waiting for more space.
  - Implies that there *is* a maximum number of elements in a queue
- The consuming thread *must block* upon receiving from an empty queue
- Blocks are NOT to be done with polling (+ sleeps), *why?*
- What should we do then? - ***Conditionals***

What is the structure of the information to pass around?

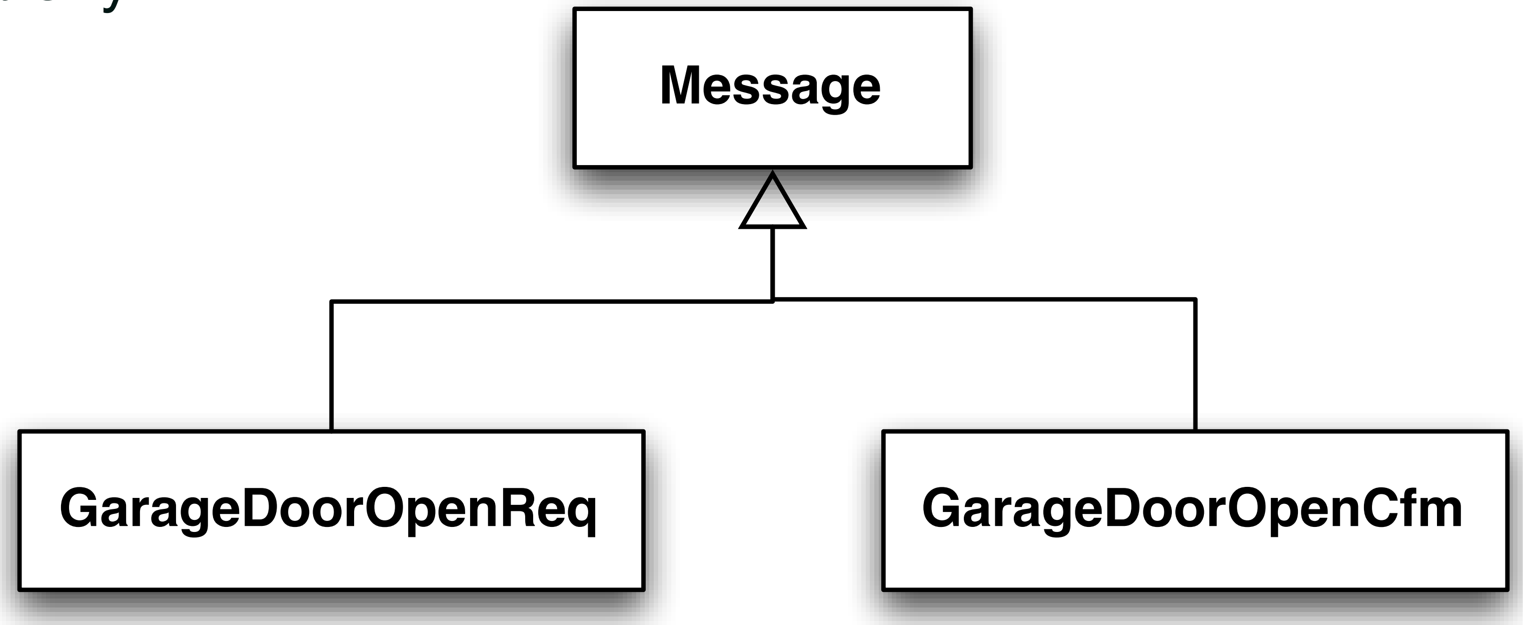
- void* or simple array of bytes
  - Can contain anything
  - No type information - No type-safety (if we don’t know what it is - we don’t know how to delete

- template based
  - Depends on the implementation, is a good solution but more complex
  - Type-safety

- Inheritance
  - Simple and extended via sub-classing
  - Type-safety / Type information - Delete via base pointer
  - Might incur overhead

Inheritance - our choice

- Message hierarchy



Message

GarageDoorOpenReq

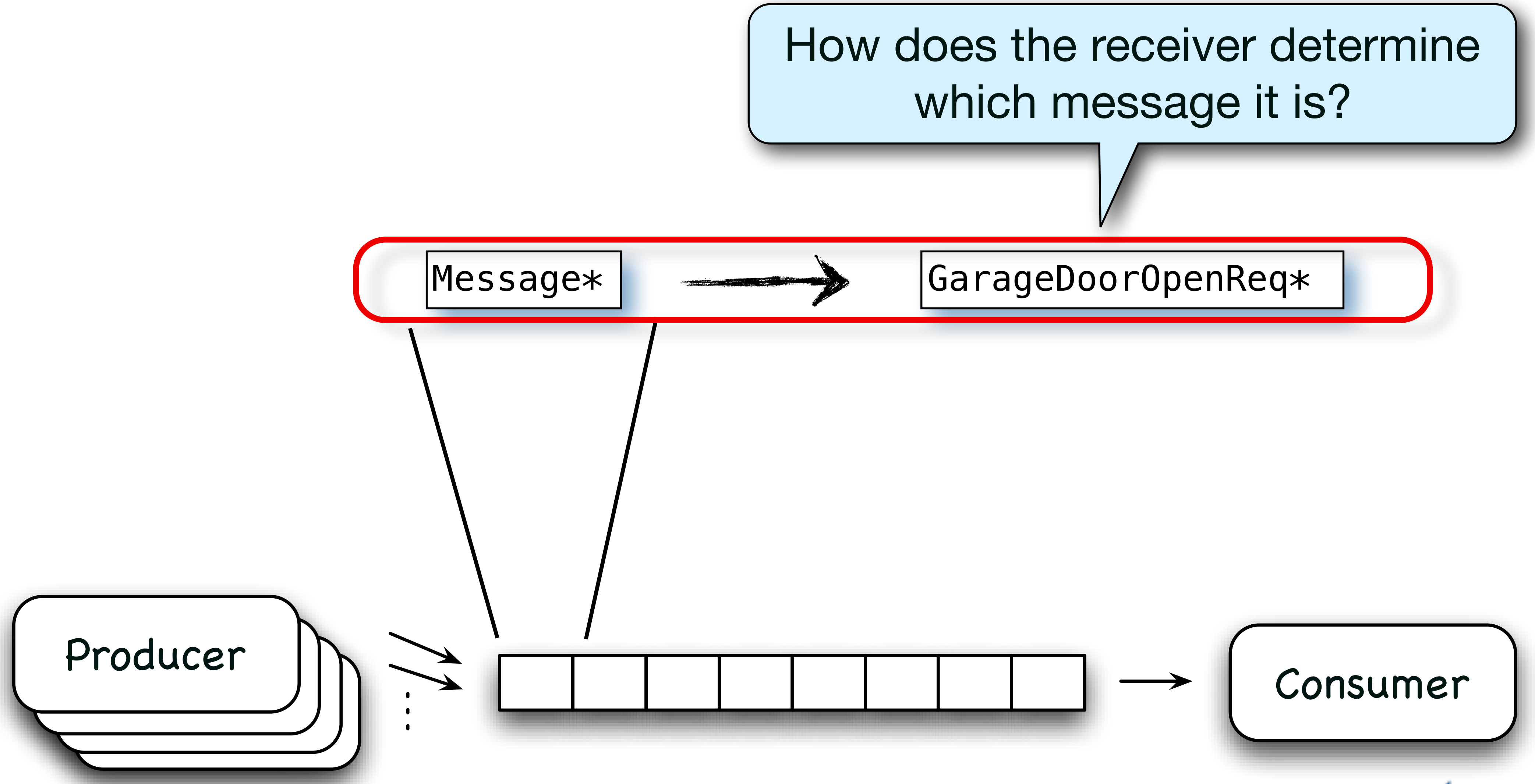
GarageDoorOpenCfm

```
class Message
{
public:
   virtual ~Message(){}
};
```

```
struct GarageDoorOpenReq : public Message
{
   MsgQueue* mq_;
};
```

Message Parsing

- A producer creates and “sends” a GarageDoorOpenReq message
  - class GarageDoorOpenReq is therefore seen as a message



How does the receiver determine which message it is?

From parent to child

- How do we convert a *Message** to a *GarageOpenDoorReq*?*
  - Via using dynamic_cast<>

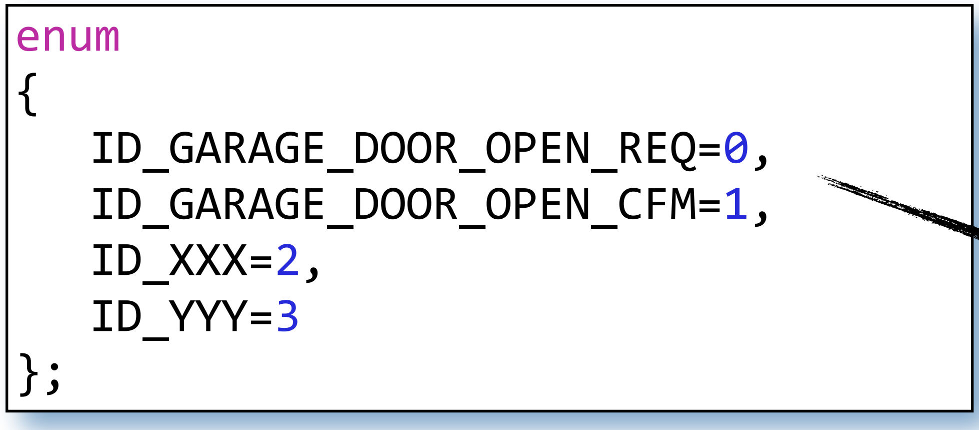
```cpp
GarageDoorOpenReq gdor;
Message* msg_ = &gdor; // Illustration!

GarageDoorOpenReq* req = dynamic_cast<GarageDoorOpenReq*>(msg_);
// Runtime check, req == NULL if not correct
```

  - Via typeid()

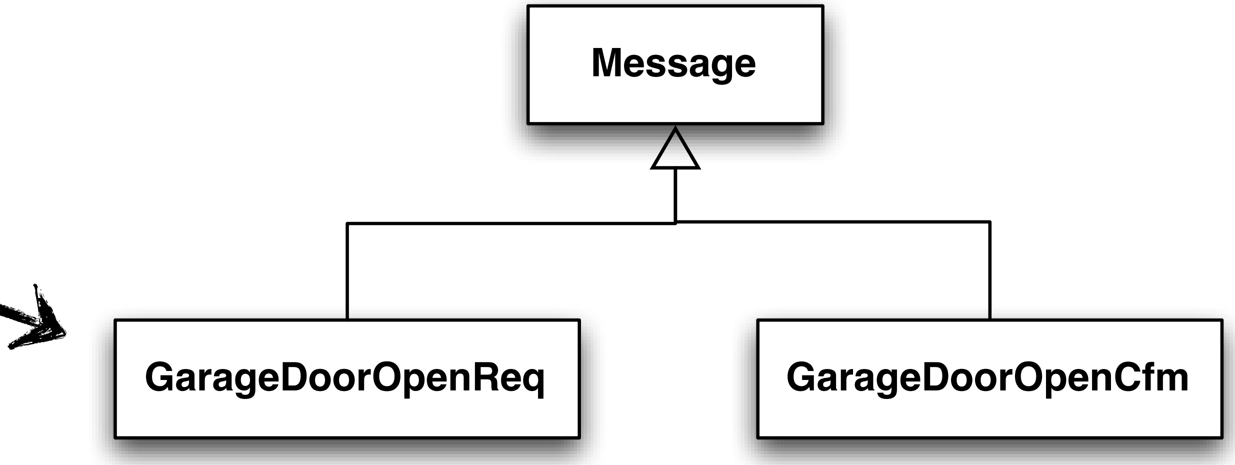
```cpp
GarageDoorOpenReq gdor;
Message* msg_ = &gdor; // Illustration!

if(typeid(*msg_) == typeid(GarageDoorOpenReq))
{
    // Runtime check - evaluates to true if pointer is of said type
    GarageDoorOpenReq* req = static_cast<GarageDoorOpenReq*>(msg_);
}
```

From parent to child

- How do we convert a *Message** to a *GarageOpenDoorReq*?*
  - Using a special identifier
    - associating an id with the message



```
enum
{
   ID_GARAGE_DOOR_OPEN_REQ=0,
   ID_GARAGE_DOOR_OPEN_CFM=1,
   ID_XXX=2,
   ID_YYY=3
};
```



Considerations regarding Embedded Systems

Embedded Compiler configurations

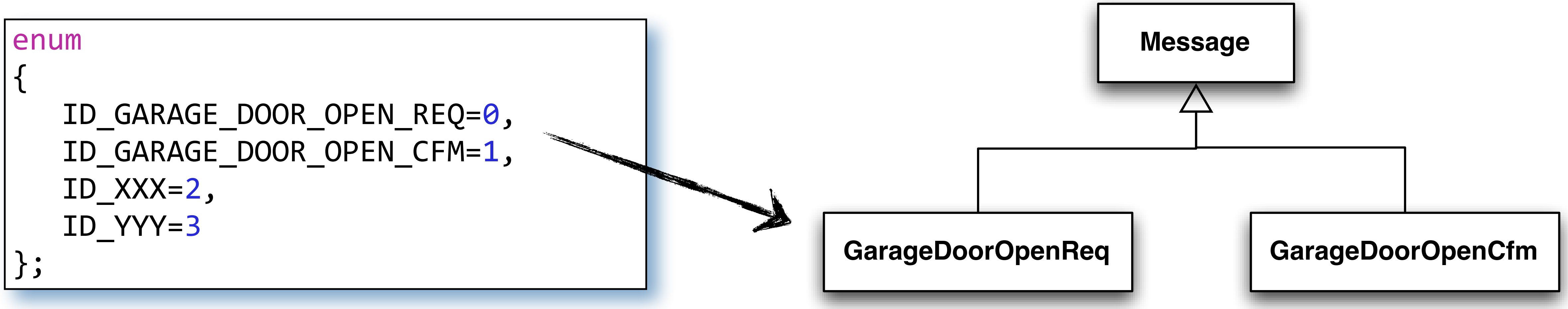
- However certain embedded compilers are compiled without support for RTTI and exception.
  - RTTI - Run Time Type Information
    - Costs in the form of space - *Yes it costs, but what are the consequences?*
  - Exceptions
    - The perception is:
      - Costs in the form of space - *What would the code handling normal errors costs?*
      - It is difficult to do correctly - *Thats certainly correct, but it is not impossible*
      - Errors are not tolerated at all, they must all be found - ***That is*** If you have the time and money, depends on the amount money

Embedded Compiler configurations

- Based on on these inputs the following requirement is added:
  - *It is acknowledged that the use of RTTI will improve program readability, however due to the increase in code size it is denounced*
    - Meaning no use of: (in our design)
      - dynamic_cast<> - Runtime check whether the cast is permissible or not
      - typeid() - Uniquely identify a given object

Due to compiler considerations

- We will be associating an id with the message



```
enum
{
   ID_GARAGE_DOOR_OPEN_REQ=0,
   ID_GARAGE_DOOR_OPEN_CFM=1,
   ID_XXX=2,
   ID_YYY=3
};
```

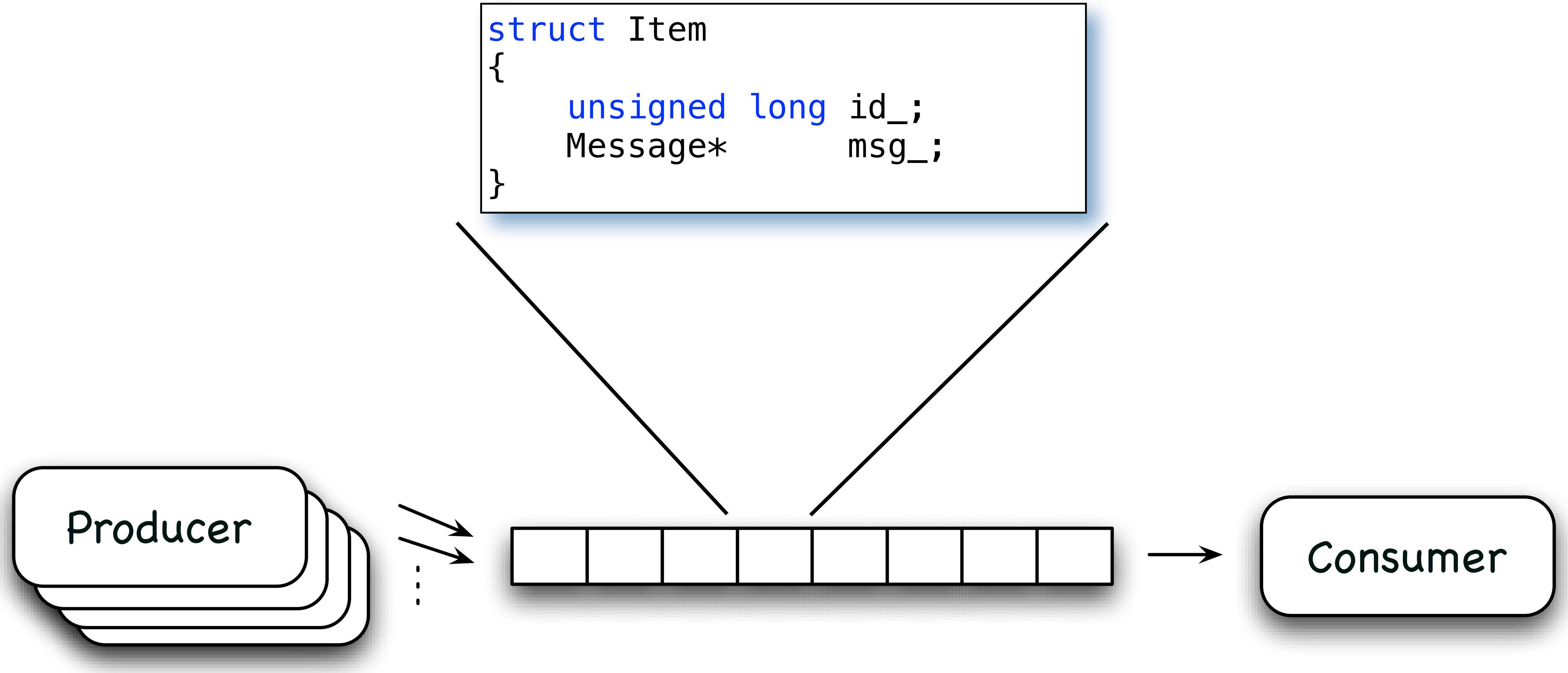
Message

GarageDoorOpenReq

GarageDoorOpenCfm

Choice of item in MsgQueue

- id_ is the identifier which is to be send
- msg_ is the message to be passed

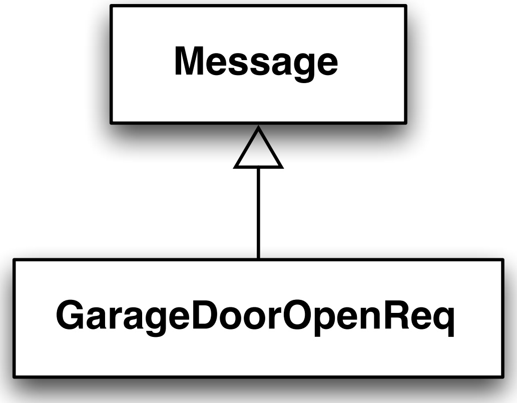


```
struct Item
{
    unsigned long id_;
    Message*      msg_;
}
```

An identifier to designate which child it is (the handler)

```
struct Item
{
    unsigned long id_;
    Message*      msg_;
}
```

```
enum
{
   ID_GARAGE_DOOR_OPEN_REQ=0,
   ID_GARAGE_DOOR_OPEN_CFM=1,
   ID_XXX=2,
   ID_YYY=3
};
```

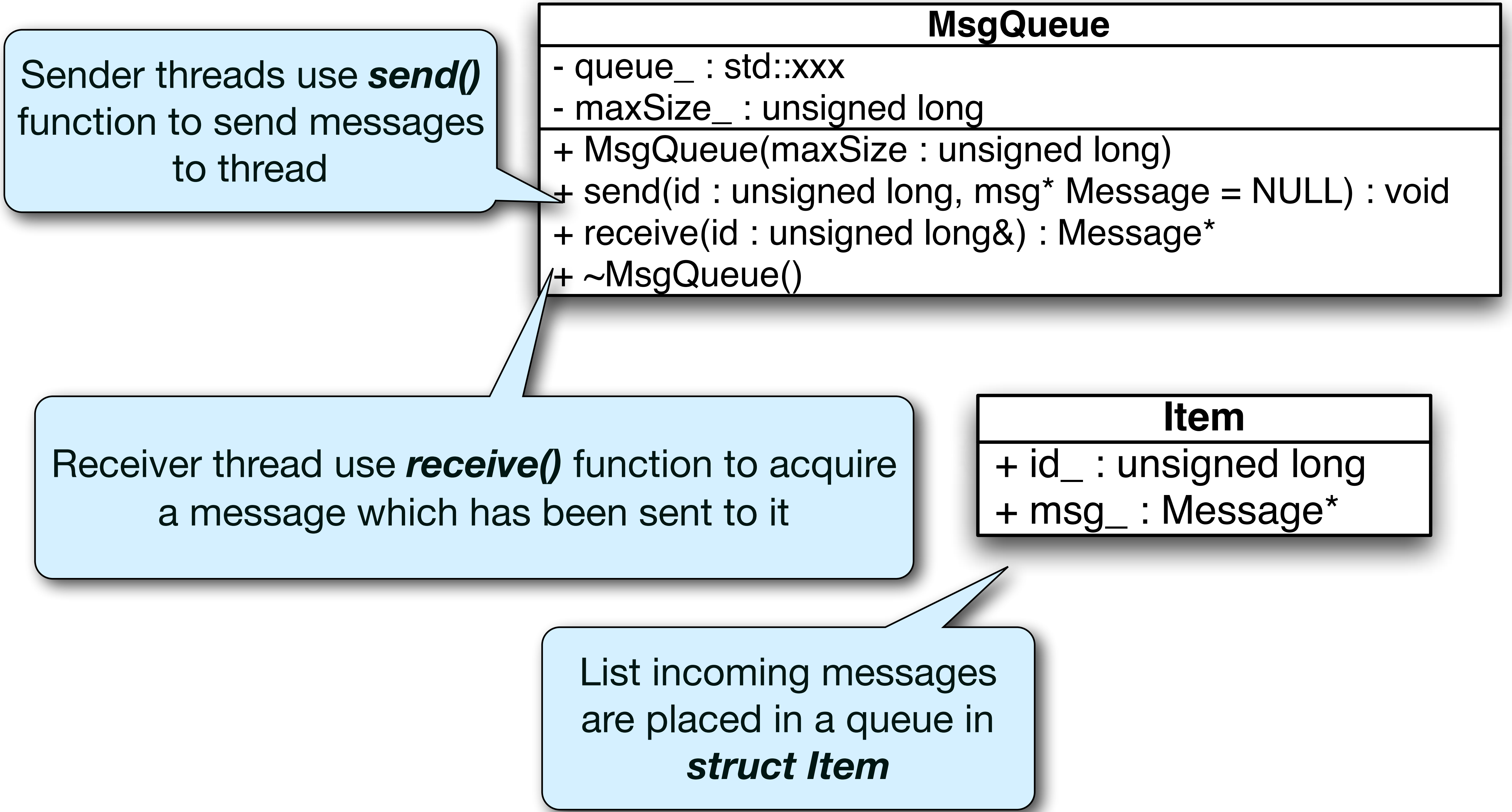


```
void handler(Message* msg, unsigned long id)
{
   switch(id)
   {
    case ID_GARAGE_DOOR_OPEN_REQ:
        GarageDoorOpenReq* gdor = static_cast<GarageDoorOpenReq*>(Msg);
        // Do stuff - call handler
        break;

    case ID_XXX:
        // ...
        break;

    default:
        std::cout << "Argh, unknown identifier, what to do???" << std::endl;
   };
}
```

Message / ID combo

- Associate an identifier with a class/structure
  - The compound signifies the control/data information to be send/received
  - The identifier is denoted by the receiving party NOT part of a globally defined enum; ***why not? Placed in a central place everyone knows; seems very good…?!***

The desired MsgQueue interface design



MsgQueue

- queue_ : std::xxx
- maxSize_ : unsigned long

+ MsgQueue(maxSize : unsigned long)
+ send(id : unsigned long, msg* Message = NULL) : void
+ receive(id : unsigned long&) : Message*
+ ~MsgQueue()

Item

+ id_ : unsigned long
+ msg_ : Message*

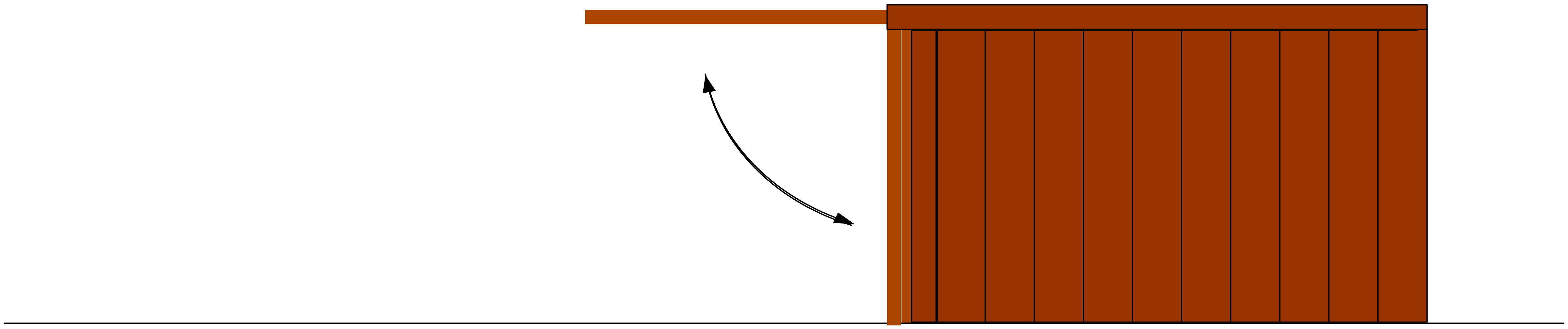
Sender threads use ***send()*** function to send messages to thread

Receiver thread use ***receive()*** function to acquire a message which has been sent to it

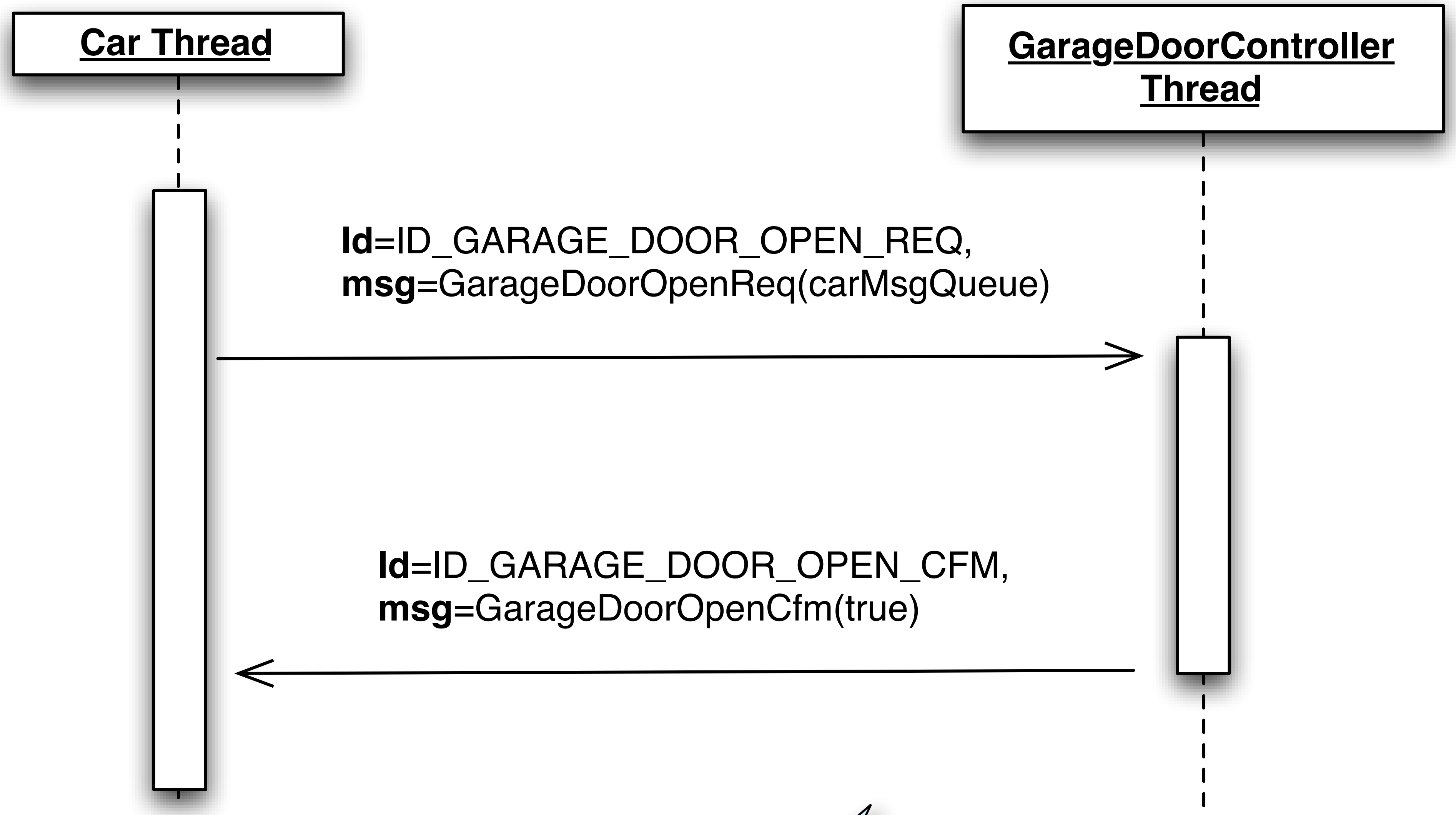
List incoming messages are placed in a queue in ***struct Item***

Case - Park-a-lot 2000

- Example: Park-a-lot 2000: An automated car parking system
  - One thread steers the car
  - Another thread steers the garage door opener

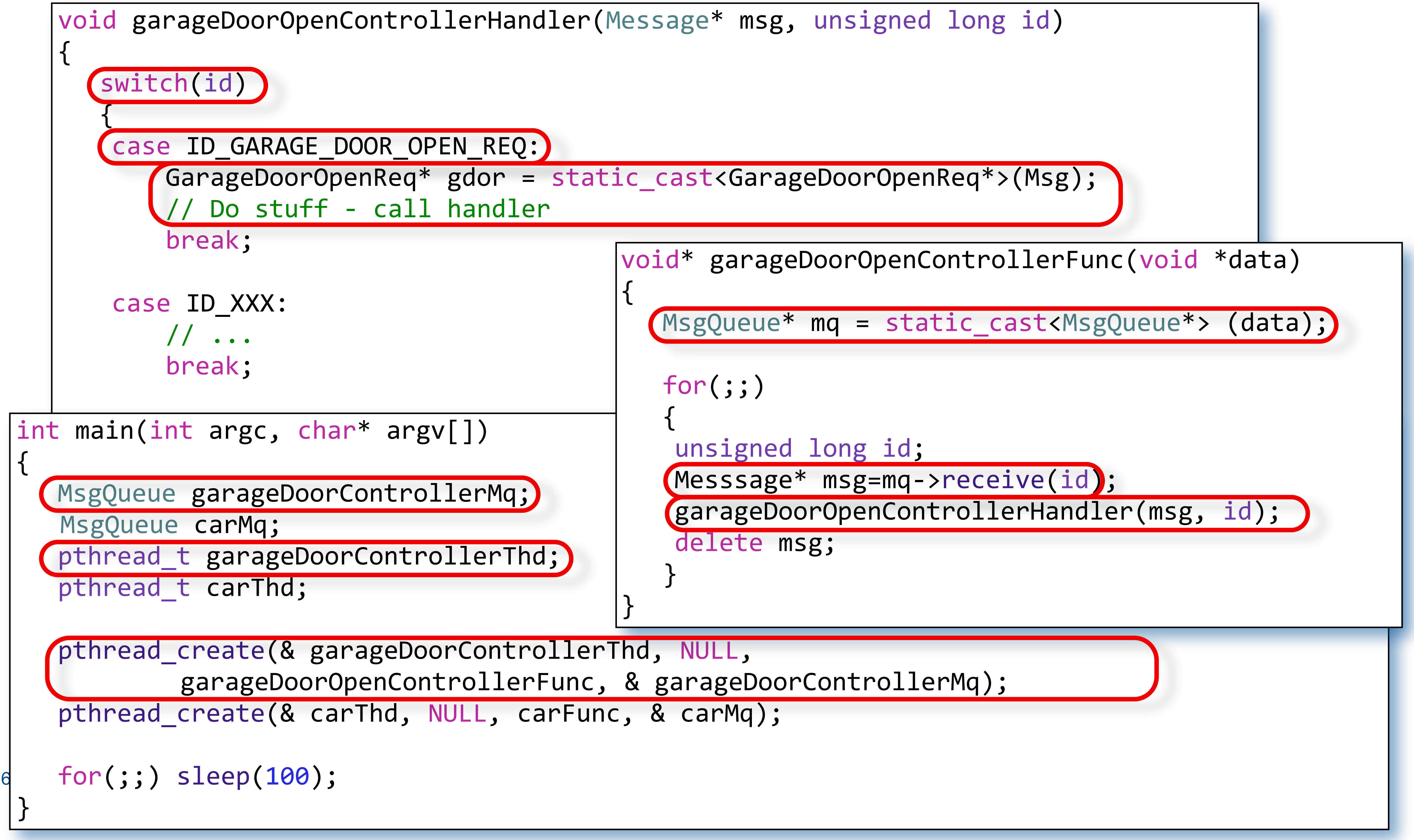


Sequence Diagram


Car Thread
GarageDoorController Thread
Id=ID_GARAGE_DOOR_OPEN_REQ,
msg=GarageDoorOpenReq(carMsgQueue)
Id=ID_GARAGE_DOOR_OPEN_CFM,
msg=GarageDoorOpenCfm(true)

3 types of signals
1. Request - Named XXXReq (a request requires a confirm)
2. Confirm - Named XXXCfm
3. Indication - Named XXXInd (purely oneway)

More complete example



```
void garageDoorOpenControllerHandler(Message* msg, unsigned long id)
{
  switch(id)
   {
   case ID_GARAGE_DOOR_OPEN_REQ:
       GarageDoorOpenReq* gdor = static_cast<GarageDoorOpenReq*>(Msg);
       // Do stuff - call handler
       break;

   case ID_XXX:
       // ...
       break;
```

```
void* garageDoorOpenControllerFunc(void *data)
{
  MsgQueue* mq = static_cast<MsgQueue*> (data);

   for(;;)
   {
    unsigned long id;
    Messsage* msg=mq->receive(id);
    garageDoorOpenControllerHandler(msg, id);
    delete msg;
   }
}
```

```
int main(int argc, char* argv[])
{
   MsgQueue garageDoorControllerMq;
   MsgQueue carMq;
   pthread_t garageDoorControllerThd;
   pthread_t carThd;

   pthread_create(& garageDoorControllerThd, NULL,
           garageDoorOpenControllerFunc, & garageDoorControllerMq);
   pthread_create(& carThd, NULL, carFunc, & carMq);

   for(;;) sleep(100);
}
```

Park-a-lot 2000 Communication

```
class Message
{
public:
   virtual ~Message(){}
};
```

Car Thread

```
void carSendingOpenReq()
{
   // Create request
   GarageDoorOpenReq* req = new GarageDoorOpenReq;
   req->mq_ = &carMq; // Who the requester is

   // Send it
   garageDoorControllerMq.send(ID_GARAGE_DOOR_OPEN_REQ, req);
}
```

```
struct GarageDoorOpenReq : public
{
   MsgQueue* mq_;
};
```

GDC Thread

```
void handleGarageOpenDoorReg(GarageDoorOpenReq* req)
{
   // Create responds
   GarageDoorOpenCfm* cfm = new GarageDoorOpenCfm;
   cfm->result_ = openGarageDoor(); // The door is open

   // Send responds to requester...
   req->mq_->send(ID_GARAGE_DOOR_OPEN_CFM, cfm);
}
```

```
struct GarageDoorOpenCfm : publi
{
   bool result_;
};
```

Car Thread

```
void handleCarOpenDoorCfm(GarageDoorOpenCfm* cfm)
{
   // Check responds
   if(cfm->result_)
   {
    driveIntoParkingLot();
   }
}
```

Typical task structure in message-based system

```
void handler()
{
   while(running_)
   {
    // get message from message queue
    switch (on state) {
       case ST_IDLE:

           switch (on message) {
               case ID_MSG:
                  // Handle message.
                  break;
               default:
                  break;
           }

           break;
       default:
           break;
    }
   }

}
```

Perform setup here that does not belong in constructor

Get a message, e.g. msgQueue->receive()

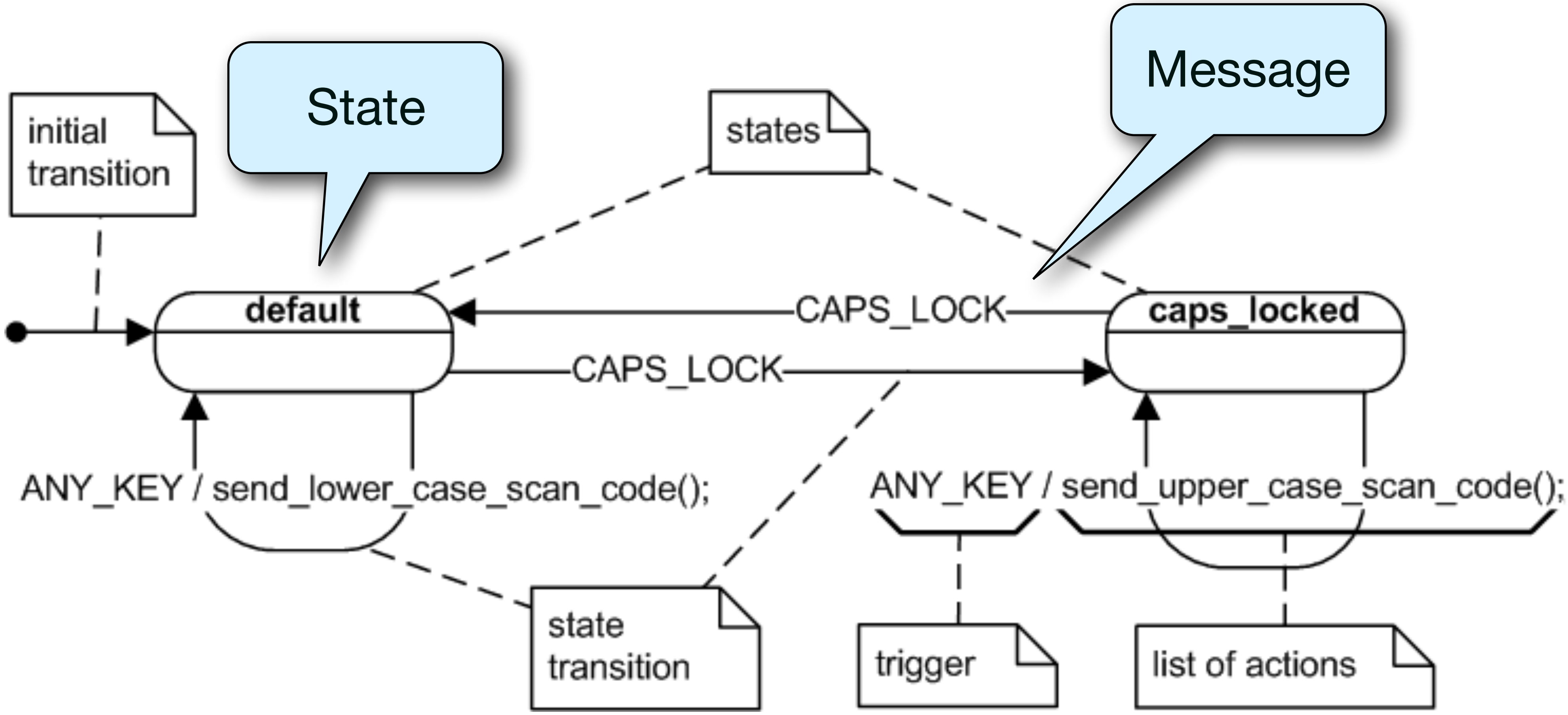
Switch on state, each state has separate handlers

Handle each ”type” of message separately

Signal design error: Thread received something it did not expect

Execution should never reach this point

Example of a State Machine



- Checkout UML Statechart at http://en.wikipedia.org/wiki/UML_state_machine

Consequences

- Negative
  - No silver bullet by far.
  - In a performance perspective not necessarily the best solution.
  - Mostly to do with a-synchronicity, meaning that you are not guaranteed an answer but have to have some form of timeout.
- Positive
  - Does not inhibit misuse, but signifies a route that makes it “more” clear, as to what is to happen when.
  - Reduces the need for critical sections e.g. mutexes and semaphores.
  - Not blocked on a conditional/mutex while waiting

Summary

- What is it we in fact have done?
    - Entered the Event Driven Programming (EDP) paradigm
- What is EDP?
    - Reaction based programming
        - Interrupts from sensors, key input, controller directives etc.
    - Multiple *correct* paths through the code
    - For more complex code structure where the code is *not* stateless state machines are the solution - *Finite State Machine (FSM)*