DUNE Developer’s Helper

of sorts.

Topics covered:

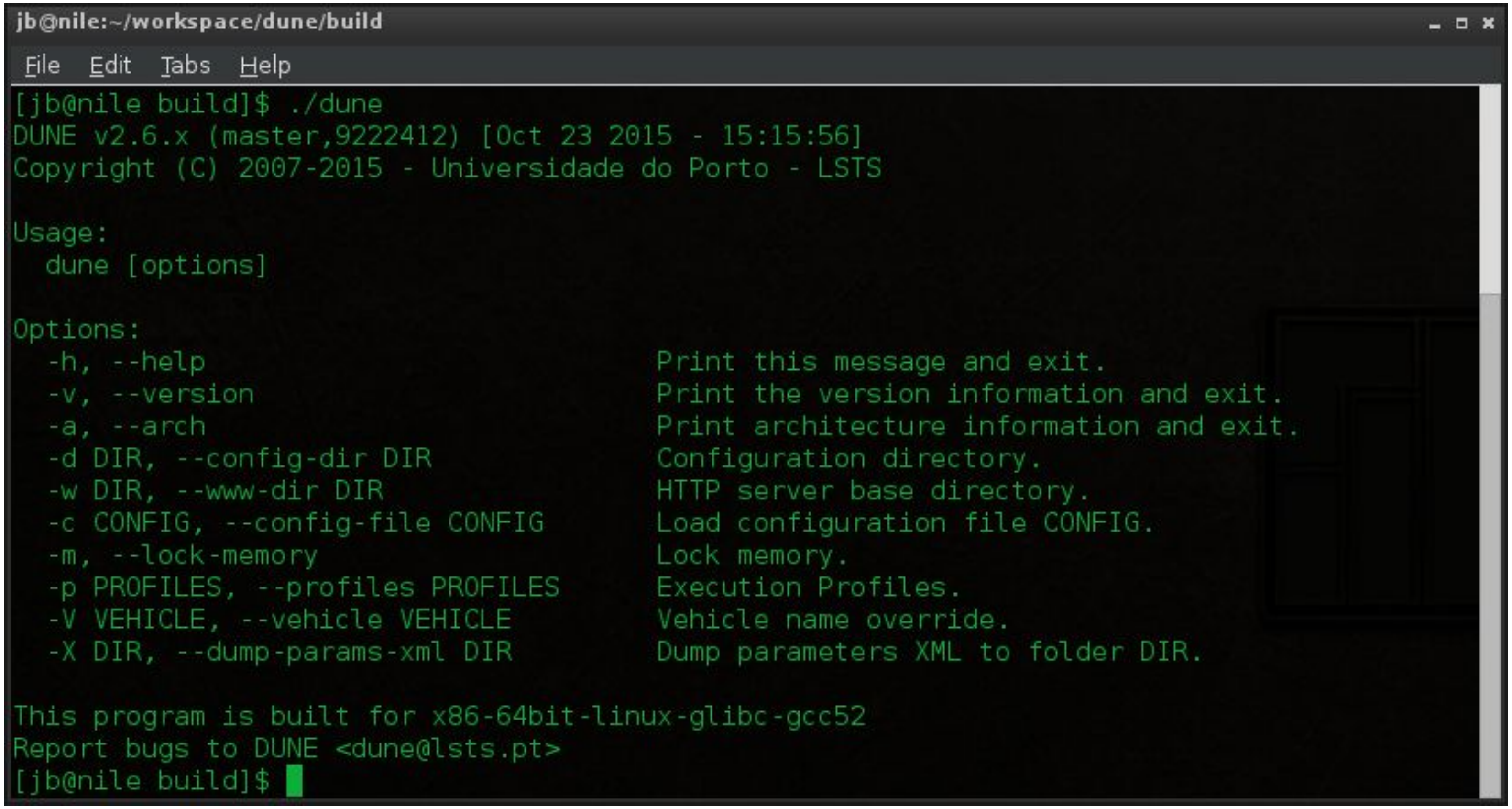
- Introduction
- Tasks creation, management / threads
- IMC
    - subscribe
    - dispatch
    - receive
    - definition
- Control architecture
- Simulation & Replay
- DUNE core functions
- Periodic
- Entity State
- Output Messages

- I18N
- Doxygen
- Git
- Style Guide

DUNE: Uniform Navigational Environment

- For Embedded Systems
- C++
- Runs everywhere except wavys and fish tags (or in autopilots!)
- Tasks: isolated, dedicated threads, that do something and (hopefully) well
- Communication amongst tasks is achieved with IMC API
- What it does:
    - comms: TCP, UDP, acoustic modems, iridium, gsm
    - logging
    - integrates sensors, actuators and power devices
    - estimation filter(s)
    - controllers - from lower level (direct actuation) to higher level

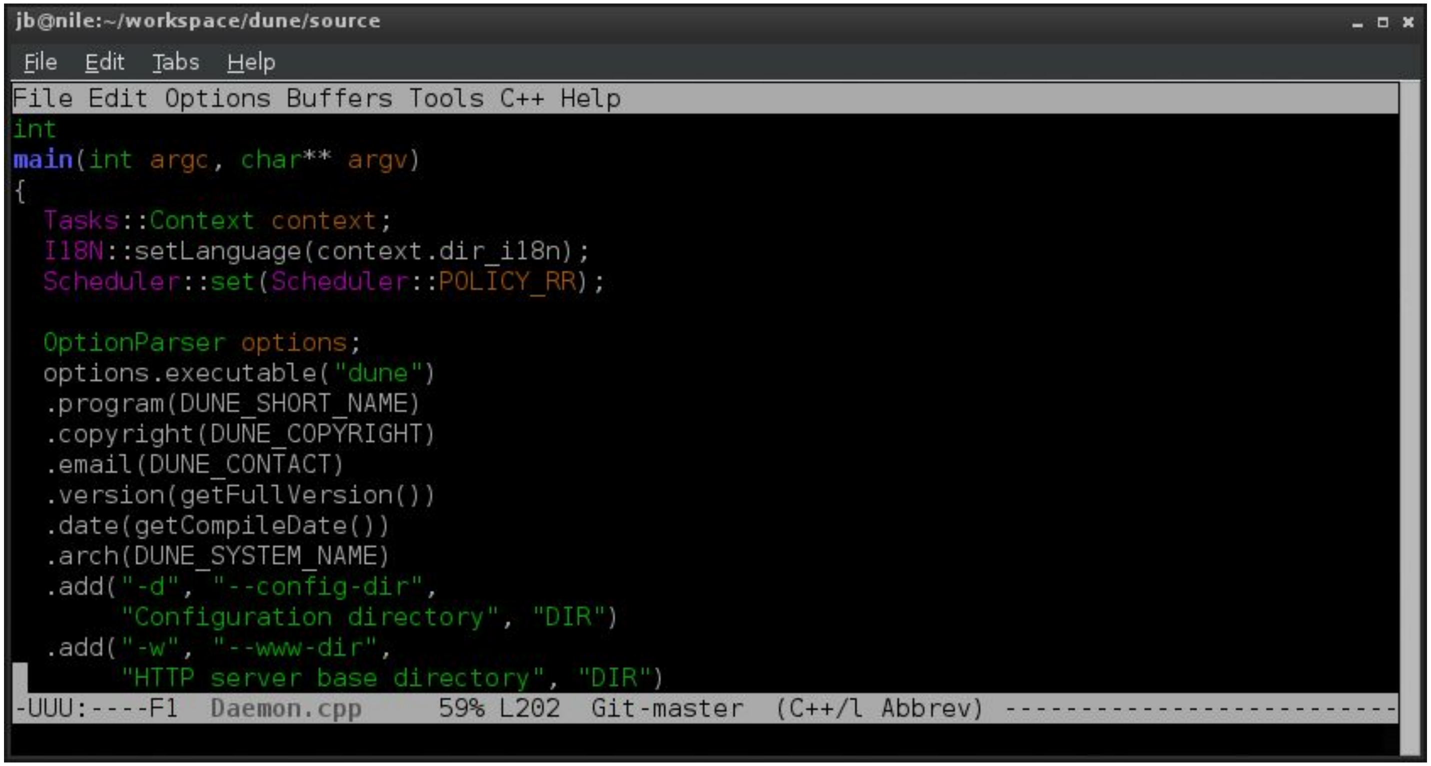
Where does it start ?!



```
[jb@nile build]$ ./dune
DUNE v2.6.x (master,9222412) [Oct 23 2015 - 15:15:56]
Copyright (C) 2007-2015 - Universidade do Porto - LSTS

Usage:
  dune [options]

Options:
  -h, --help                            Print this message and exit.
  -v, --version                         Print the version information and exit.
  -a, --arch                            Print architecture information and exit.
  -d DIR, --config-dir DIR              Configuration directory.
  -w DIR, --www-dir DIR                 HTTP server base directory.
  -c CONFIG, --config-file CONFIG       Load configuration file CONFIG.
  -m, --lock-memory                     Lock memory.
  -p PROFILES, --profiles PROFILES      Execution Profiles.
  -V VEHICLE, --vehicle VEHICLE         Vehicle name override.
  -X DIR, --dump-params-xml DIR         Dump parameters XML to folder DIR.

This program is built for x86-64bit-linux-glibc-gcc52
Report bugs to DUNE <dune@lsts.pt>
[jb@nile build]$
```

src/Main/Daemon.cpp

DUNE::Daemon **daemon**(context, options.value("--profiles"));


jb@nile:~/workspace/dune/source
File Edit Tabs Help
File Edit Options Buffers Tools C++ Help
-UUU:----F1 Daemon.cpp   59% L202 Git-master (C++/l Abbrev) -------------------------

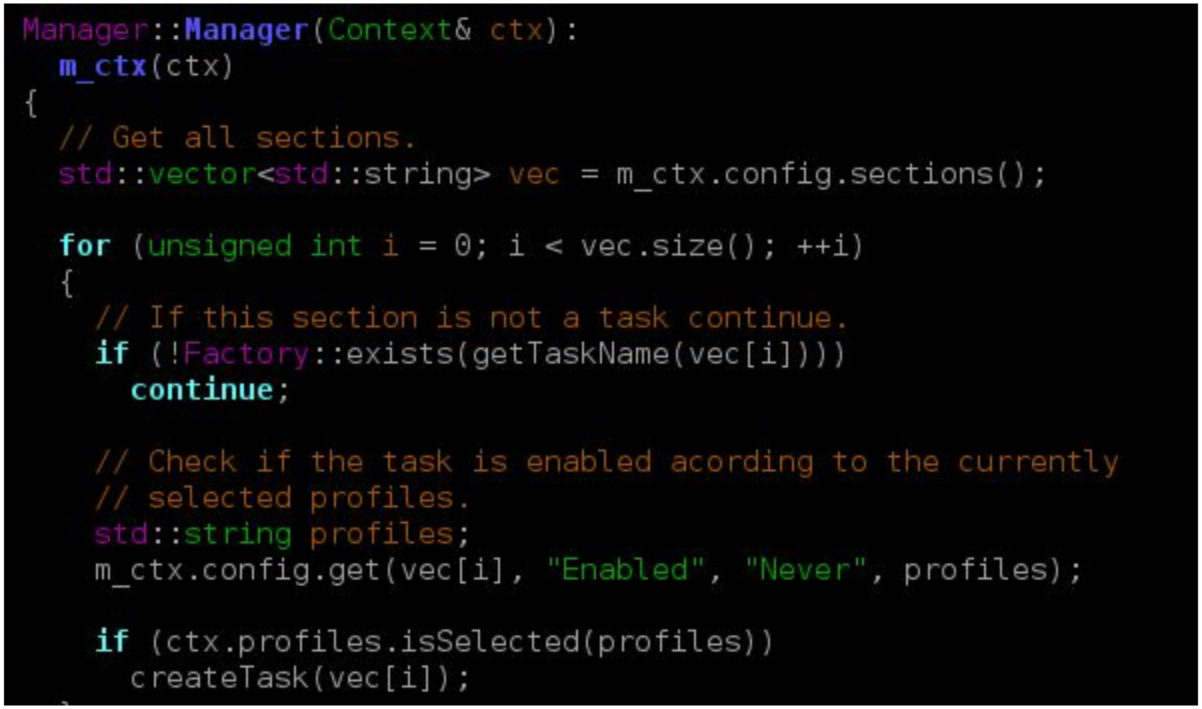
```
int
main(int argc, char** argv)
{
  Tasks::Context context;
  I18N::setLanguage(context.dir_i18n);
  Scheduler::set(Scheduler::POLICY_RR);

  OptionParser options;
  options.executable("dune")
  .program(DUNE_SHORT_NAME)
  .copyright(DUNE_COPYRIGHT)
  .email(DUNE_CONTACT)
  .version(getFullVersion())
  .date(getCompileDate())
  .arch(DUNE_SYSTEM_NAME)
  .add("-d", "--config-dir",
       "Configuration directory", "DIR")
  .add("-w", "--www-dir",
       "HTTP server base directory", "DIR")
```

src/DUNE/Daemon.cpp

```
Tasks::Manager* m_tman;

m_tman = new DUNE::Tasks::Manager(m_ctx);
m_tman->start();
```

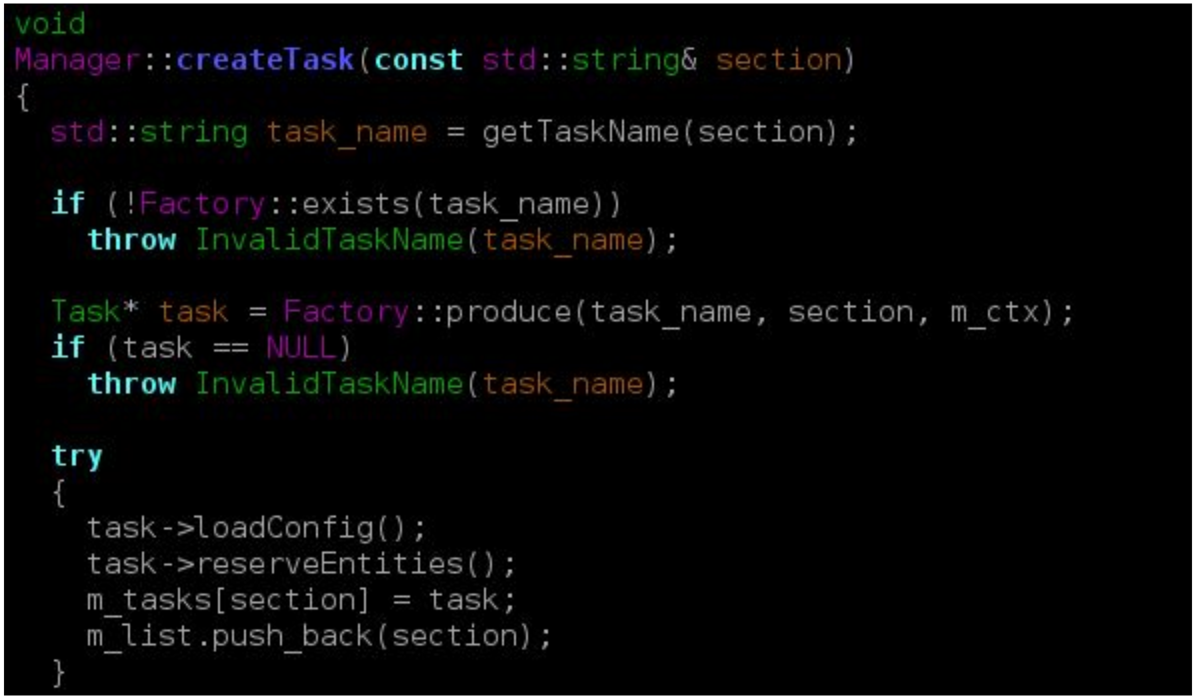
src/DUNE/Tasks/Manager.?pp

```
//! Running tasks.
std::map<std::string, Task*> m_tasks;
```



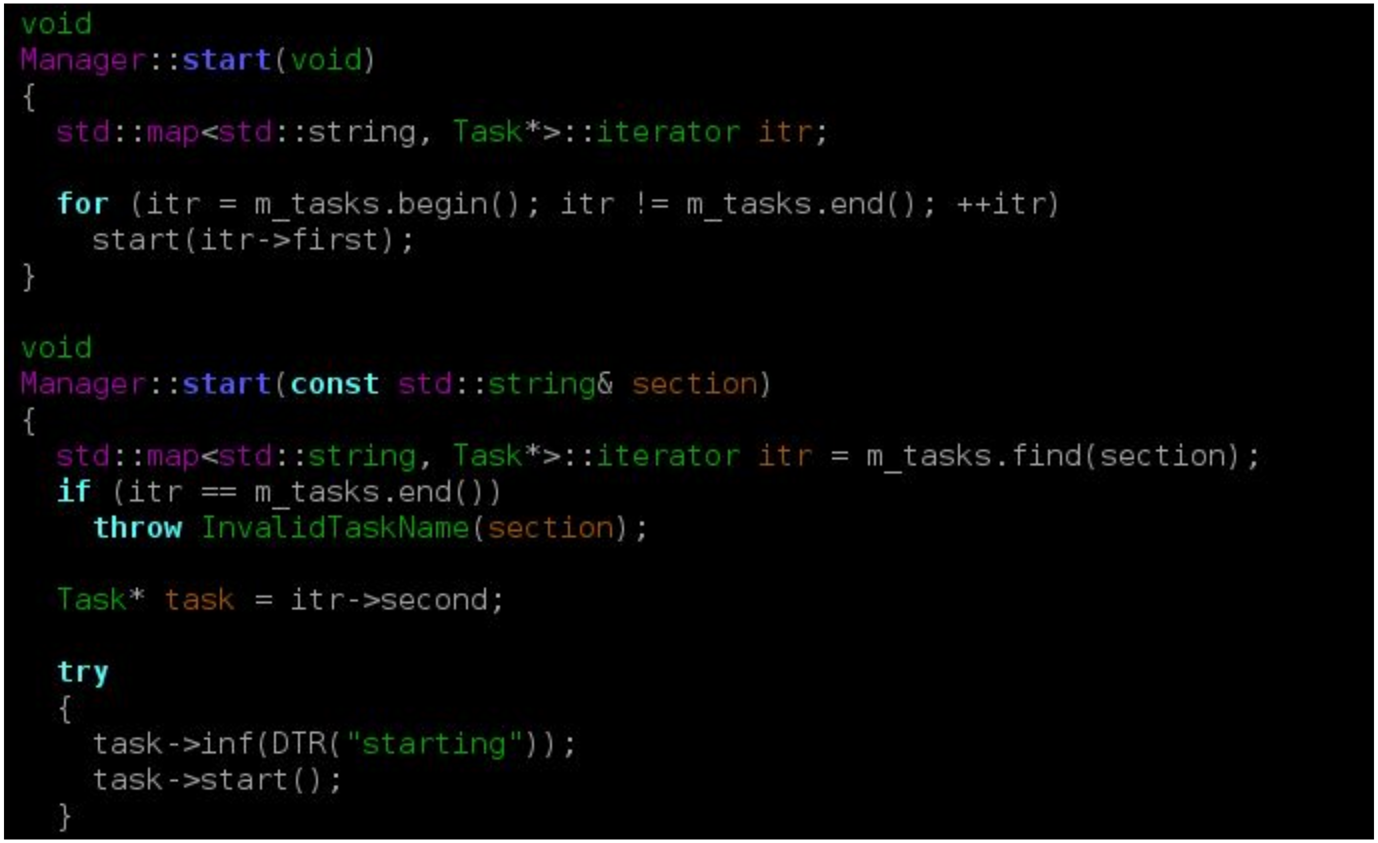
```
Manager::Manager(Context& ctx):
  m_ctx(ctx)
{
  // Get all sections.
  std::vector<std::string> vec = m_ctx.config.sections();

  for (unsigned int i = 0; i < vec.size(); ++i)
  {
    // If this section is not a task continue.
    if (!Factory::exists(getTaskName(vec[i])))
      continue;

    // Check if the task is enabled acording to the currently
    // selected profiles.
    std::string profiles;
    m_ctx.config.get(vec[i], "Enabled", "Never", profiles);

    if (ctx.profiles.isSelected(profiles))
      createTask(vec[i]);
  }
```

src/DUNE/Tasks/Manager.?pp



```
void
Manager::createTask(const std::string& section)
{
  std::string task_name = getTaskName(section);

  if (!Factory::exists(task_name))
    throw InvalidTaskName(task_name);

  Task* task = Factory::produce(task_name, section, m_ctx);
  if (task == NULL)
    throw InvalidTaskName(task_name);

  try
  {
    task->loadConfig();
    task->reserveEntities();
    m_tasks[section] = task;
    m_list.push_back(section);
  }
```

src/DUNE/Tasks/Manager.?pp

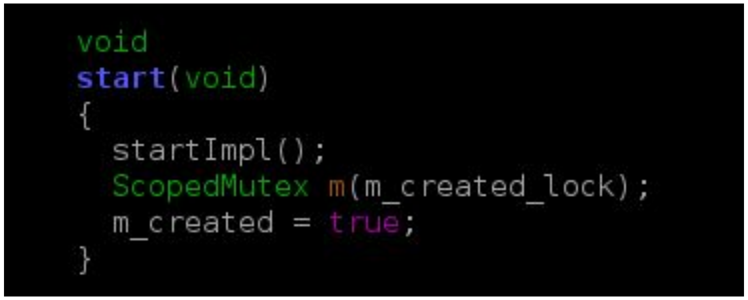


```cpp
void
Manager::start(void)
{
  std::map<std::string, Task*>::iterator itr;

  for (itr = m_tasks.begin(); itr != m_tasks.end(); ++itr)
    start(itr->first);
}

void
Manager::start(const std::string& section)
{
  std::map<std::string, Task*>::iterator itr = m_tasks.find(section);
  if (itr == m_tasks.end())
    throw InvalidTaskName(section);

  Task* task = itr->second;

  try
  {
    task->inf(DTR("starting"));
    task->start();
  }
```

src/DUNE/Tasks/Task.?pp

DUNE::Tasks::Task.hpp

class Task: **public** AbstractTask
class AbstractTask **public** Concurrency::Thread
class Thread **public** Runnable

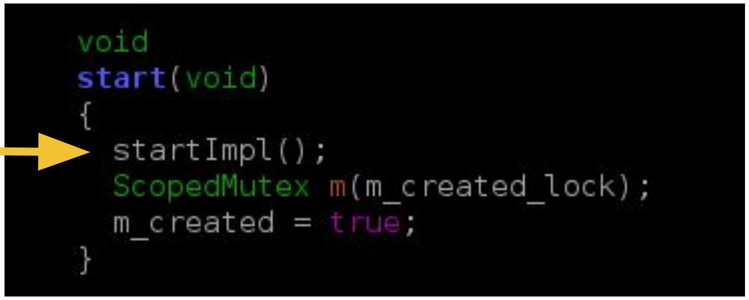


```
void
start(void)
{
  startImpl();
  ScopedMutex m(m_created_lock);
  m_created = true;
}
```

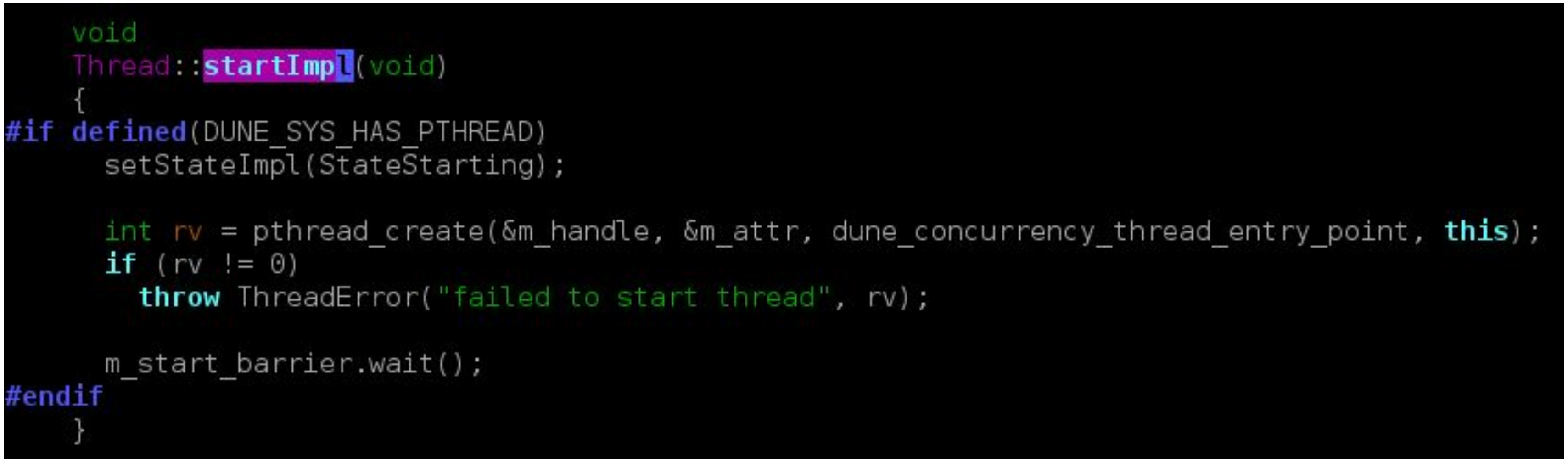
src/DUNE/Tasks/Task.?pp

DUNE::Tasks::Task.hpp

class Task: **public** AbstractTask
 class AbstractTask **public** Concurrency::Thread
  class Thread **public** Runnable



```
    void
    start(void)
    {
      startImpl();
      ScopedMutex m(m_created_lock);
      m_created = true;
    }
```



```
    void
    Thread::startImpl(void)
    {
#if defined(DUNE_SYS_HAS_PTHREAD)
      setStateImpl(StateStarting);

      int rv = pthread_create(&m_handle, &m_attr, dune_concurrency_thread_entry_point, this);
      if (rv != 0)
        throw ThreadError("failed to start thread", rv);

      m_start_barrier.wait();
#endif
    }
```

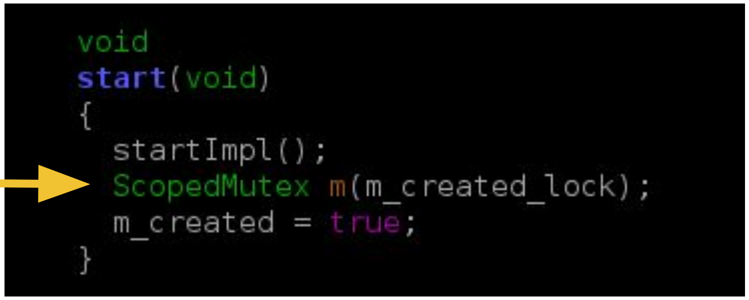
src/DUNE/Tasks/Task.?pp

DUNE::Tasks::Task.hpp

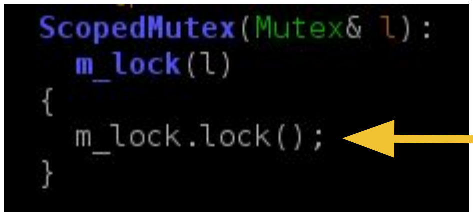
class Task: **public** AbstractTask

class AbstractTask **public** Concurrency::Thread

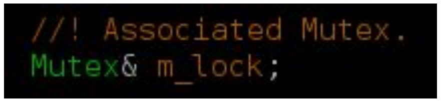
class Thread **public** Runnable



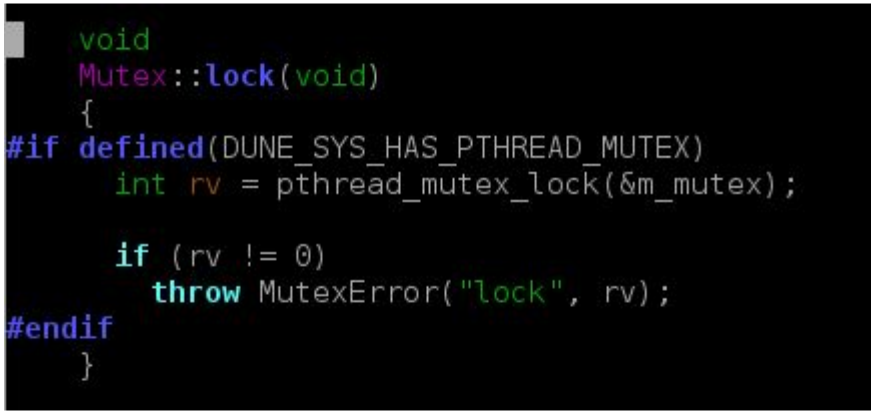
```
void
start(void)
{
  startImpl();
  ScopedMutex m(m_created_lock);
  m_created = true;
}
```



```
ScopedMutex(Mutex& l):
  m_lock(l)
{
  m_lock.lock();
}
```



```
//! Associated Mutex.
Mutex& m_lock;
```



```
    void
    Mutex::lock(void)
    {
#if defined(DUNE_SYS_HAS_PTHREAD_MUTEX)
      int rv = pthread_mutex_lock(&m_mutex);

      if (rv != 0)
        throw MutexError("lock", rv);
#endif
    }
```

Thread management

In DUNE, there are at least **N+1** threads where **N** is the number of tasks plus the Daemon thread.

Each task can launch new threads

- Database access;
- Comms: HayesModem class; BasicModem class; HTTP; IridiumSBD;
- Sensors: BlueView, Echo Sounder,
- System shutdown commands: MantaPanel, Supervisors/Power

- Parameter “Execution Priority” is an index that distinguishes threads priority (default 10). The higher, more priority it has

Thread management

The Operative System manages thread execution
Frequency is defined in a kernel parameter: **CONFIG_HZ**

In LAUV: 1000 Hz
    i.e.: at each 1 ms, the scheduler changes running thread.

Atom CPU: 1000 Hz
IGEP: 100 Hz
BBB: 250 Hz
RPI: 1000 Hz

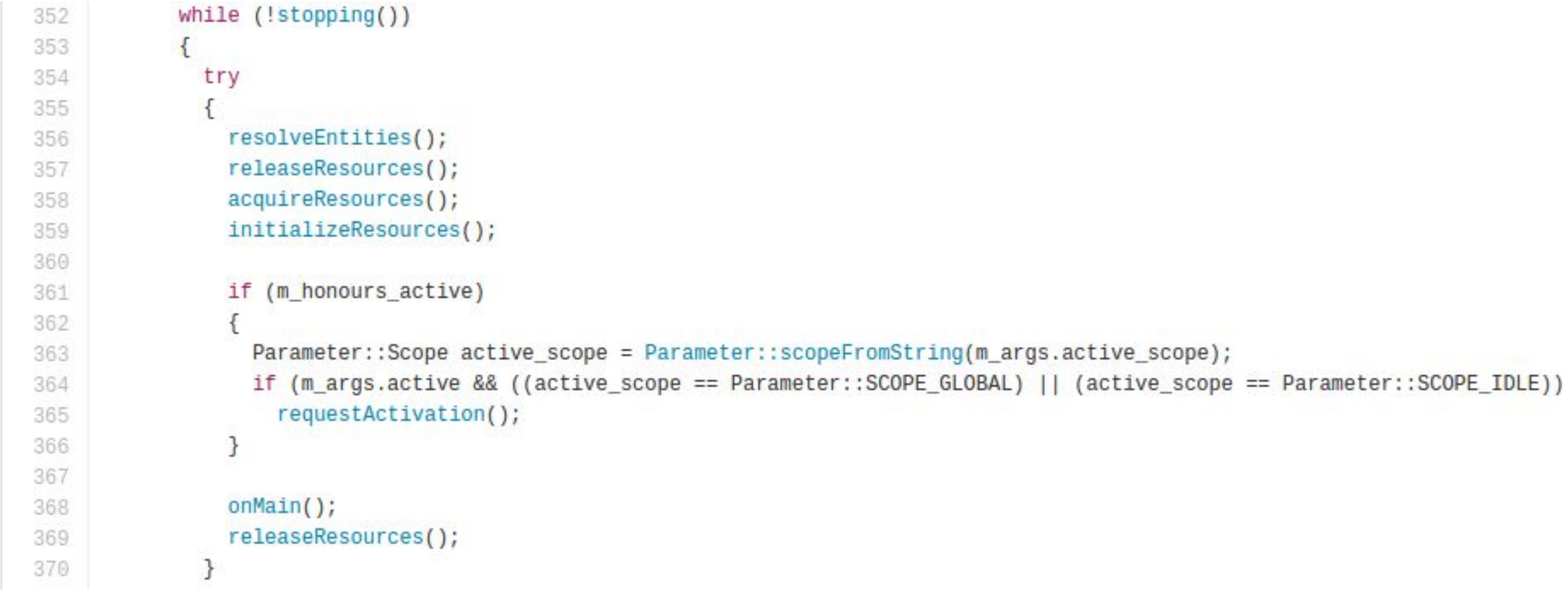
src/DUNE/Tasks/Task.?pp

Basic Functions

- void onEntityReservation(void)
  - Task can reserve additional entities, i.e. additional source entity addresses.
- void onEntityResolution(void)
  - Task can resolve entities, i.e., get the source entity address of entities using entity label (e.g: resolveEntity("IMU"))
- void onResourceAcquisition(void)
  - Task can acquire resources (open serial ports, sockets, etc), instantiate objects
- void onResourceInitialization(void)
  - Initialize previously acquired resources (e.g: run configurations)
- void onResourceRelease(void)
  - Releases all acquired resources. Runs once after entities resolution and at the end.
- void onMain(void) / void task(void)

src/DUNE/Tasks/Task.?pp

Basic Functions



```cpp
    while (!stopping())
    {
      try
      {
        resolveEntities();
        releaseResources();
        acquireResources();
        initializeResources();

        if (m_honours_active)
        {
          Parameter::Scope active_scope = Parameter::scopeFromString(m_args.active_scope);
          if (m_args.active && ((active_scope == Parameter::SCOPE_GLOBAL) || (active_scope == Parameter::SCOPE_IDLE))
            requestActivation();
        }

        onMain();
        releaseResources();
      }
```

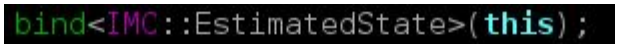
IMC: Why?

- To exchange information amongst Tasks (*)
- To log data (Data.lsf logs are stacks of IMC messages)

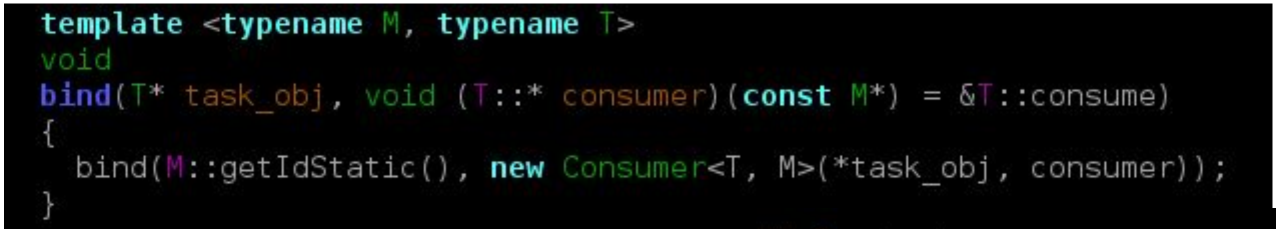
How does it work ?

(*) we also use it to send external messages to other DUNE/NEPTUS systems

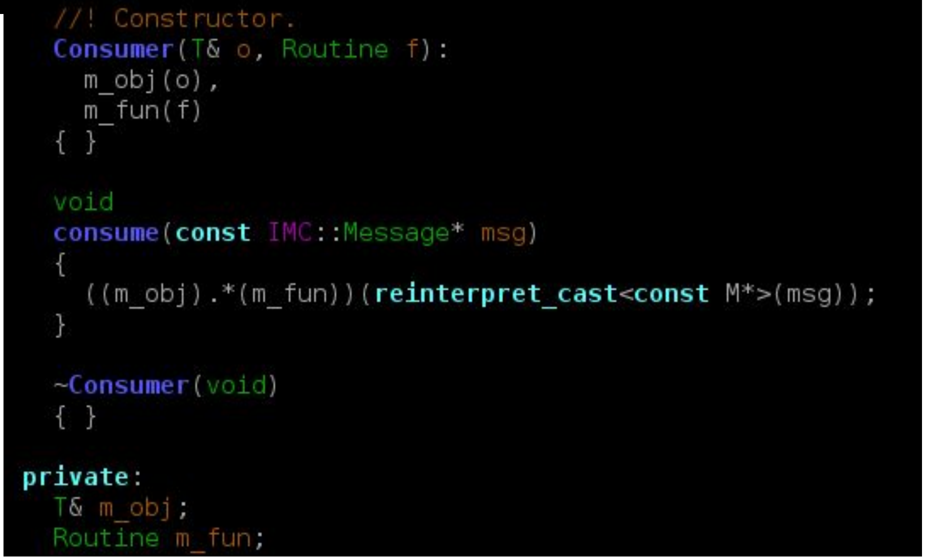
IMC: Subscribe messages: *bind*



```
bind<IMC::EstimatedState>(this);
```



```
template <typename M, typename T>
void
bind(T* task_obj, void (T::* consumer)(const M*) = &T::consume)
{
  bind(M::getIdStatic(), new Consumer<T, M>(*task_obj, consumer));
}
```

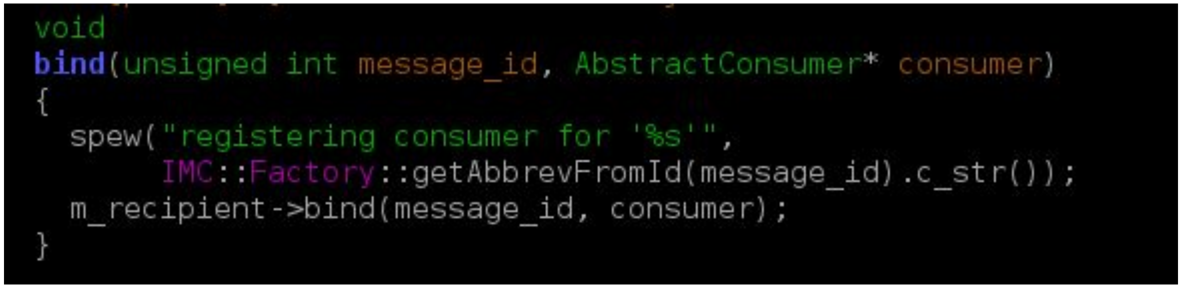


```
  //! Constructor.
  Consumer(T& o, Routine f):
    m_obj(o),
    m_fun(f)
  { }

  void
  consume(const IMC::Message* msg)
  {
    ((m_obj).*(m_fun))(reinterpret_cast<const M*>(msg));
  }

  ~Consumer(void)
  { }

private:
  T& m_obj;
  Routine m_fun;
```

With *bind* (and complement *consume* function) we are subscribing the task to a message type.

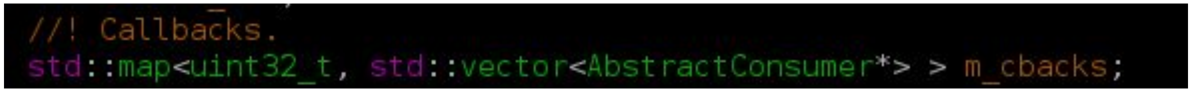
e.g: I want to receive all messages of type *EstimatedState*

IMC: Subscribe messages: *bind*

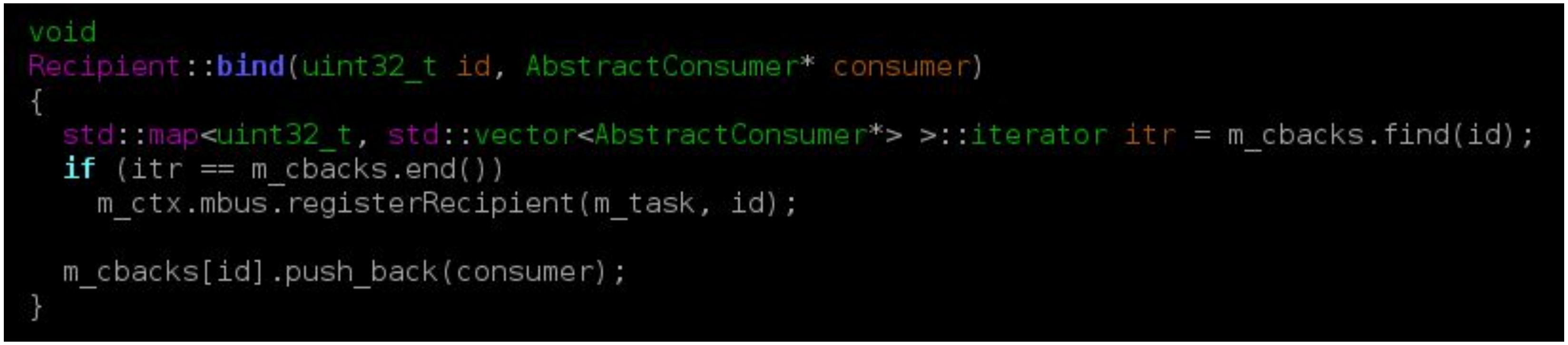


```
void
bind(unsigned int message_id, AbstractConsumer* consumer)
{
  spew("registering consumer for '%s'",
       IMC::Factory::getAbbrevFromId(message_id).c_str());
  m_recipient->bind(message_id, consumer);
}
```

Once task has subscribed to the messages it can start receiving them.



```
//! Callbacks.
std::map<uint32_t, std::vector<AbstractConsumer*> > m_cbacks;
```



```
void
Recipient::bind(uint32_t id, AbstractConsumer* consumer)
{
  std::map<uint32_t, std::vector<AbstractConsumer*> >::iterator itr = m_cbacks.find(id);
  if (itr == m_cbacks.end())
    m_ctx.mbus.registerRecipient(m_task, id);

  m_cbacks[id].push_back(consumer);
}
```

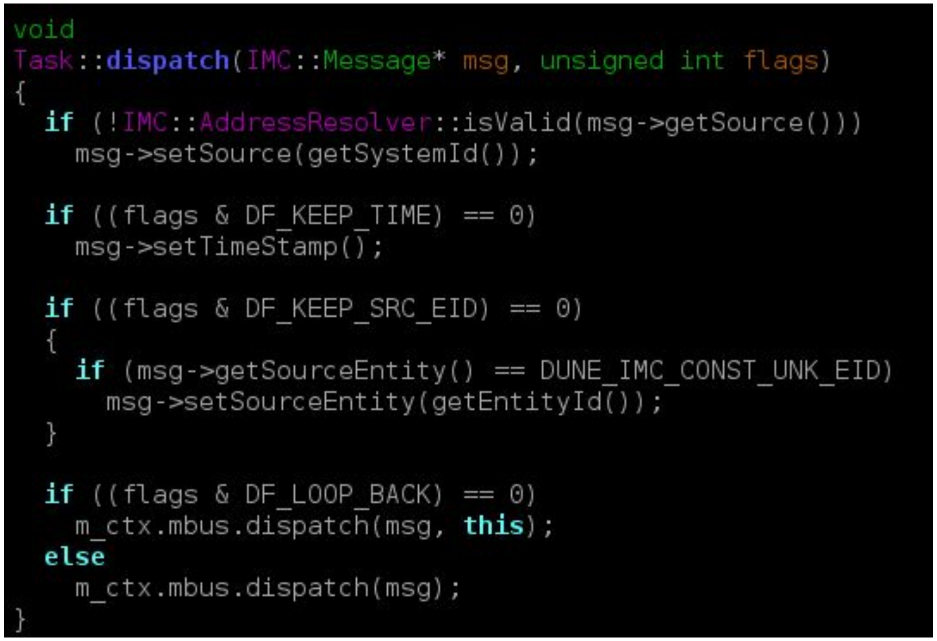
IMC: Send messages: *dispatch*

No subscription is required to send messages to the bus. Any message is accepted.

To send messages to the network, *dispatch* is used

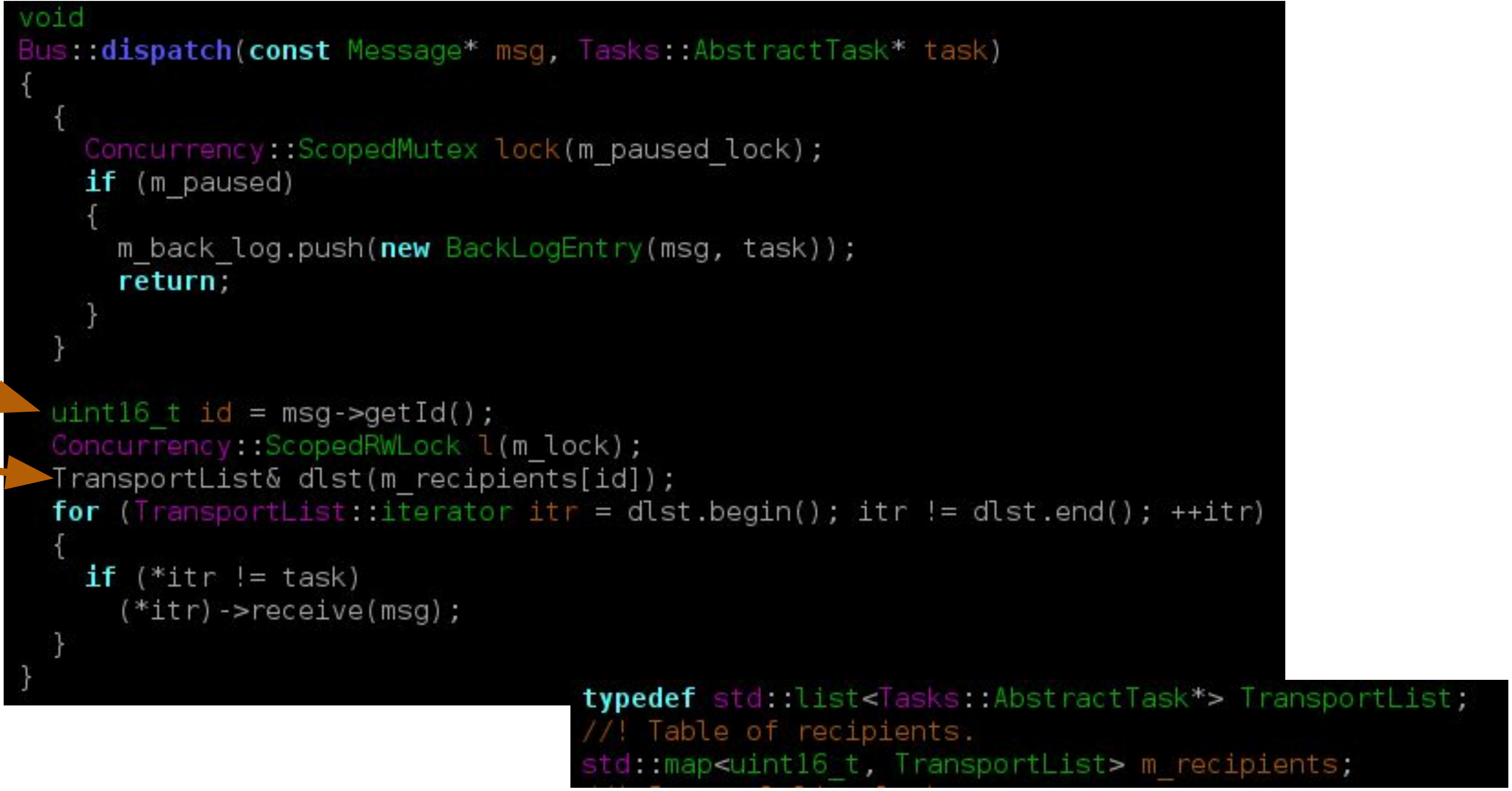
Flags:

- DF_KEEP_TIME: do not override timestamp
- DF_KEEP_SRC_EID: do not override source entity id
- DF_LOOP_BACK: loopback message to my consume



```
void
Task::dispatch(IMC::Message* msg, unsigned int flags)
{
  if (!IMC::AddressResolver::isValid(msg->getSource()))
    msg->setSource(getSystemId());

  if ((flags & DF_KEEP_TIME) == 0)
    msg->setTimeStamp();

  if ((flags & DF_KEEP_SRC_EID) == 0)
  {
    if (msg->getSourceEntity() == DUNE_IMC_CONST_UNK_EID)
      msg->setSourceEntity(getEntityId());
  }

  if ((flags & DF_LOOP_BACK) == 0)
    m_ctx.mbus.dispatch(msg, this);
  else
    m_ctx.mbus.dispatch(msg);
}
```

IMC: Send messages: *dispatch*



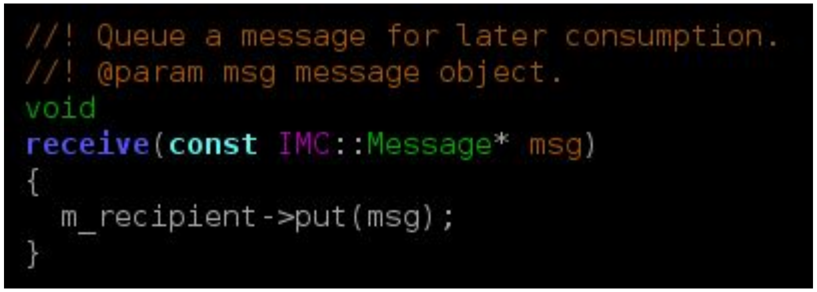
```
void
Bus::dispatch(const Message* msg, Tasks::AbstractTask* task)
{
  {
    Concurrency::ScopedMutex lock(m_paused_lock);
    if (m_paused)
    {
      m_back_log.push(new BackLogEntry(msg, task));
      return;
    }
  }

  uint16_t id = msg->getId();
  Concurrency::ScopedRWLock l(m_lock);
  TransportList& dlst(m_recipients[id]);
  for (TransportList::iterator itr = dlst.begin(); itr != dlst.end(); ++itr)
  {
    if (*itr != task)
      (*itr)->receive(msg);
  }
}
```

```
typedef std::list<Tasks::AbstractTask*> TransportList;
//! Table of recipients.
std::map<uint16_t, TransportList> m_recipients;
```

IMC
message id

Recipients
per id

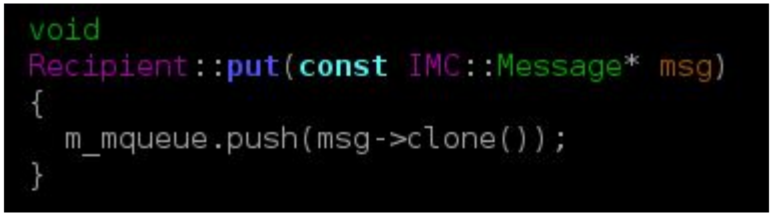
IMC: Send messages: *dispatch*



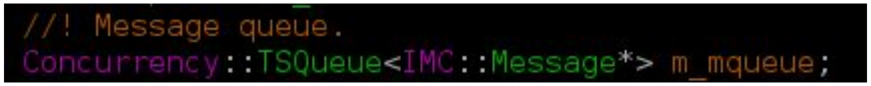
```
//! Queue a message for later consumption.
//! @param msg message object.
void
receive(const IMC::Message* msg)
{
  m_recipient->put(msg);
}
```

Message is added to a queue controlled by Recipient.

Recipient works as a mailbox where messages stay waiting to be consumed.

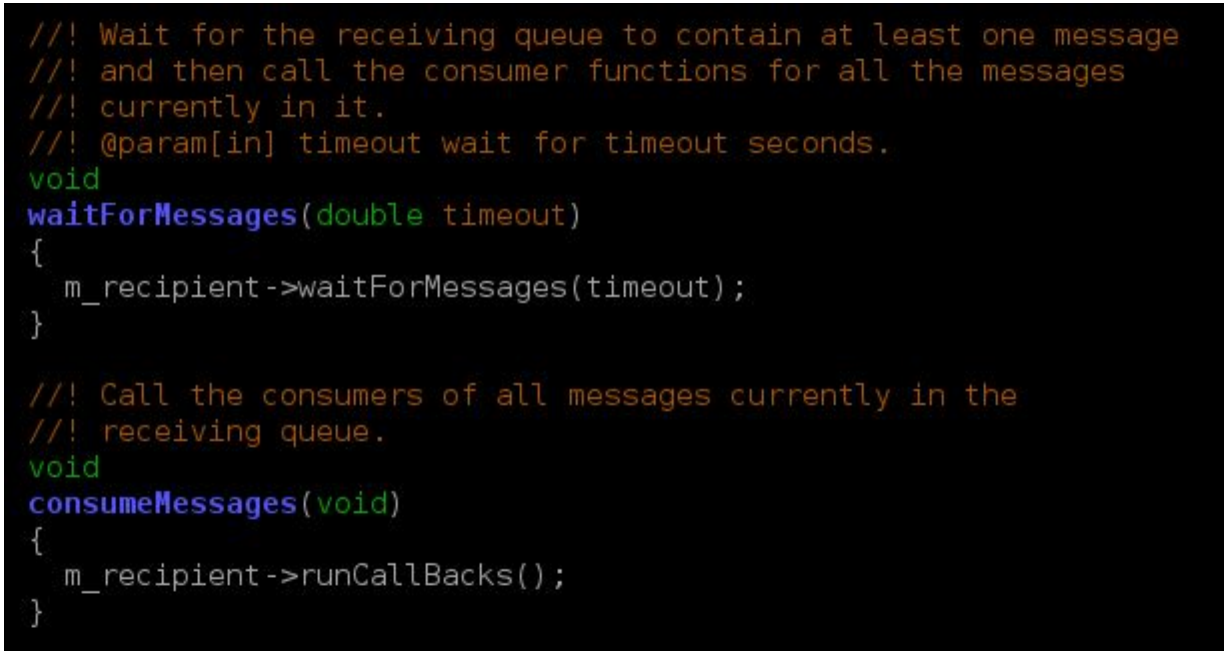


```
void
Recipient::put(const IMC::Message* msg)
{
  m_mqueue.push(msg->clone());
}
```



```
//! Message queue.
Concurrency::TSQueue<IMC::Message*> m_mqueue;
```

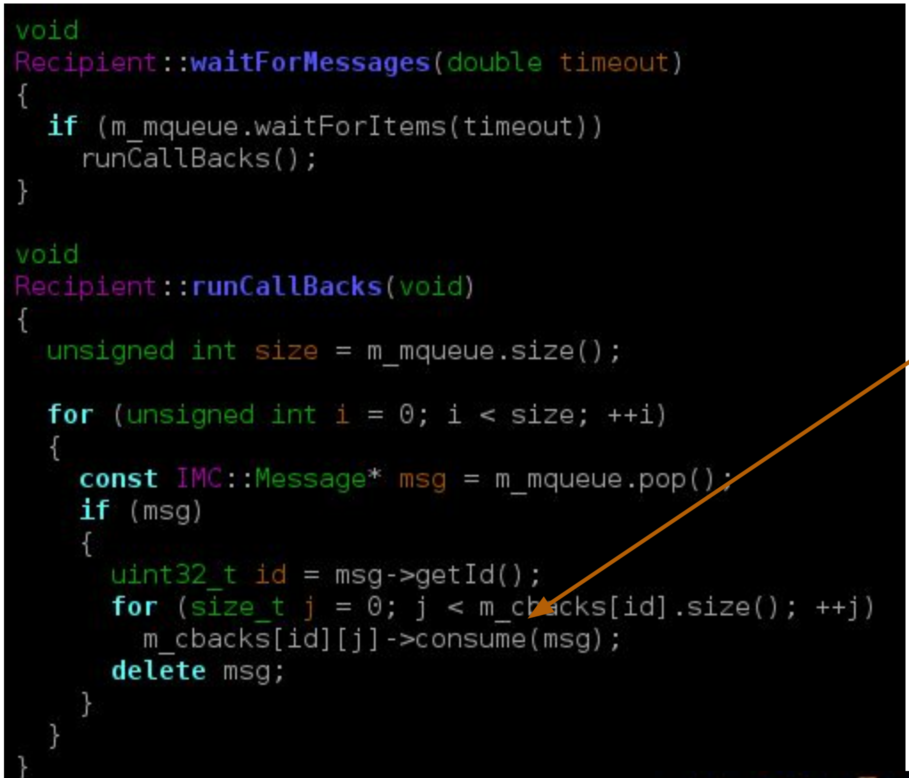
IMC: Let’s receive: *consume*



```
//! Wait for the receiving queue to contain at least one message
//! and then call the consumer functions for all the messages
//! currently in it.
//! @param[in] timeout wait for timeout seconds.
void
waitForMessages(double timeout)
{
  m_recipient->waitForMessages(timeout);
}

//! Call the consumers of all messages currently in the
//! receiving queue.
void
consumeMessages(void)
{
  m_recipient->runCallBacks();
}
```

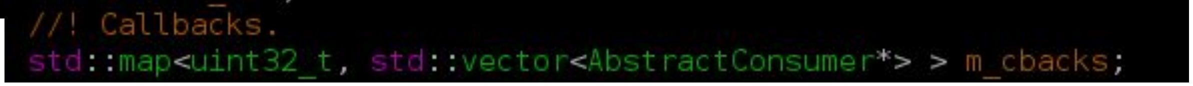
In all tasks, during onMain execution, either waitForMessages() or consumeMessages() need to be called

IMC: Let’s receive: *consume*



```
void
Recipient::waitForMessages(double timeout)
{
  if (m_mqueue.waitForItems(timeout))
    runCallBacks();
}

void
Recipient::runCallBacks(void)
{
  unsigned int size = m_mqueue.size();

  for (unsigned int i = 0; i < size; ++i)
  {
    const IMC::Message* msg = m_mqueue.pop();
    if (msg)
    {
      uint32_t id = msg->getId();
      for (size_t j = 0; j < m_cbacks[id].size(); ++j)
        m_cbacks[id][j]->consume(msg);
      delete msg;
    }
  }
}
```

Task consumes are called so that messages can be processed



```
//! Callbacks.
std::map<uint32_t, std::vector<AbstractConsumer*> > m_cbacks;
```

IMC: what is it?

Message Oriented Protocol - **not** a communication protocol, a messaging protocol

- One XML document defines all messages
- Generators for documentation, C++ and Java code
- Serialization/deserialization to/from:
  - JSON
  - XML
  - Binary
- Serialized messages are used for logging and communication
- Binary serialization format can be translated to human-readable format (LLF)

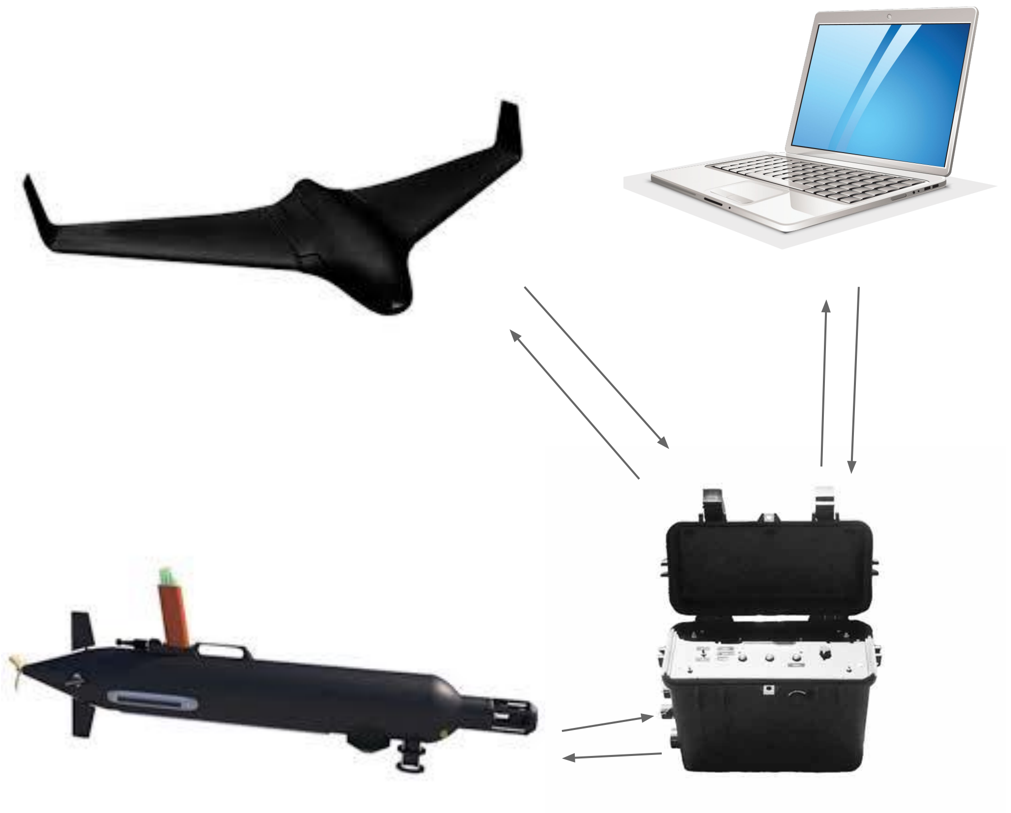
IMC: definition

Addresses are partitioned in classes (AUV, UAV, ROV, CCU, etc)

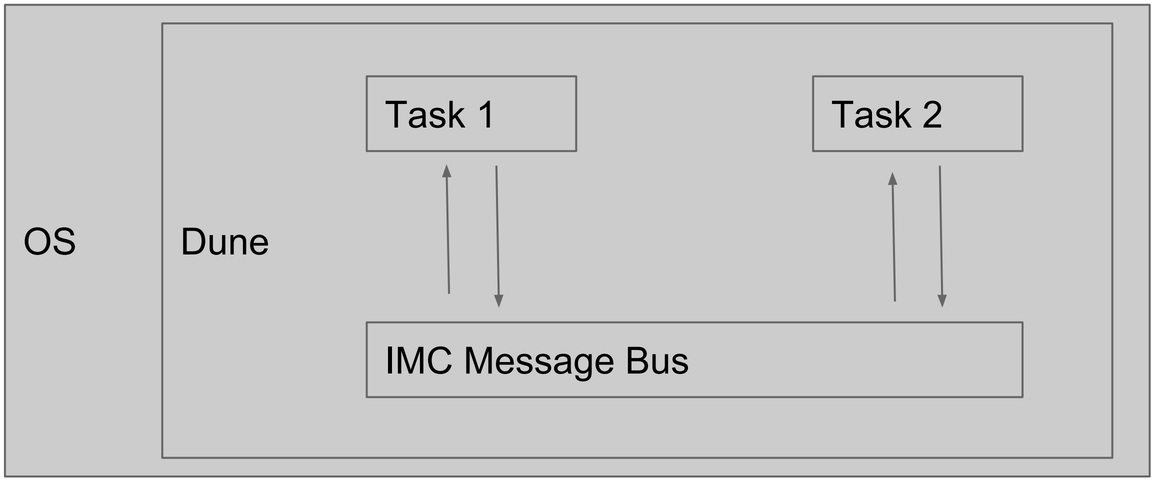
- Each system has a unique address (i.e., unique number)

- Subsystems/submodules of a system are called entities

- Each entity has a unique local number used to further qualify a message (e.g., disambiguate messages of the same type but different sources, temperature from a CTD vs CPU Temperature)



IMC: anatomy described



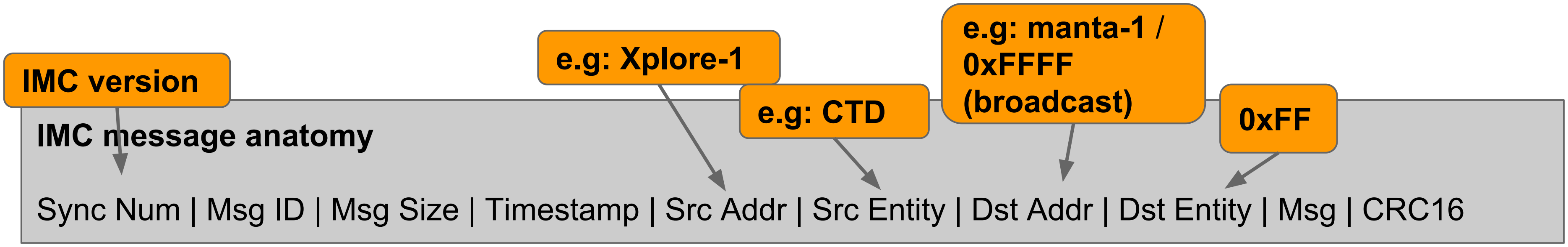
OS

Dune

Task 1

Task 2

IMC Message Bus



IMC version

e.g: Xplore-1

e.g: CTD

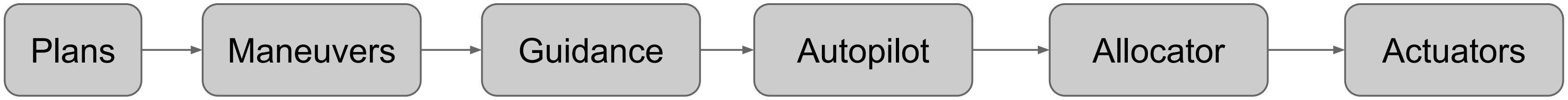
e.g: manta-1 / 0xFFFF (broadcast)

0xFF

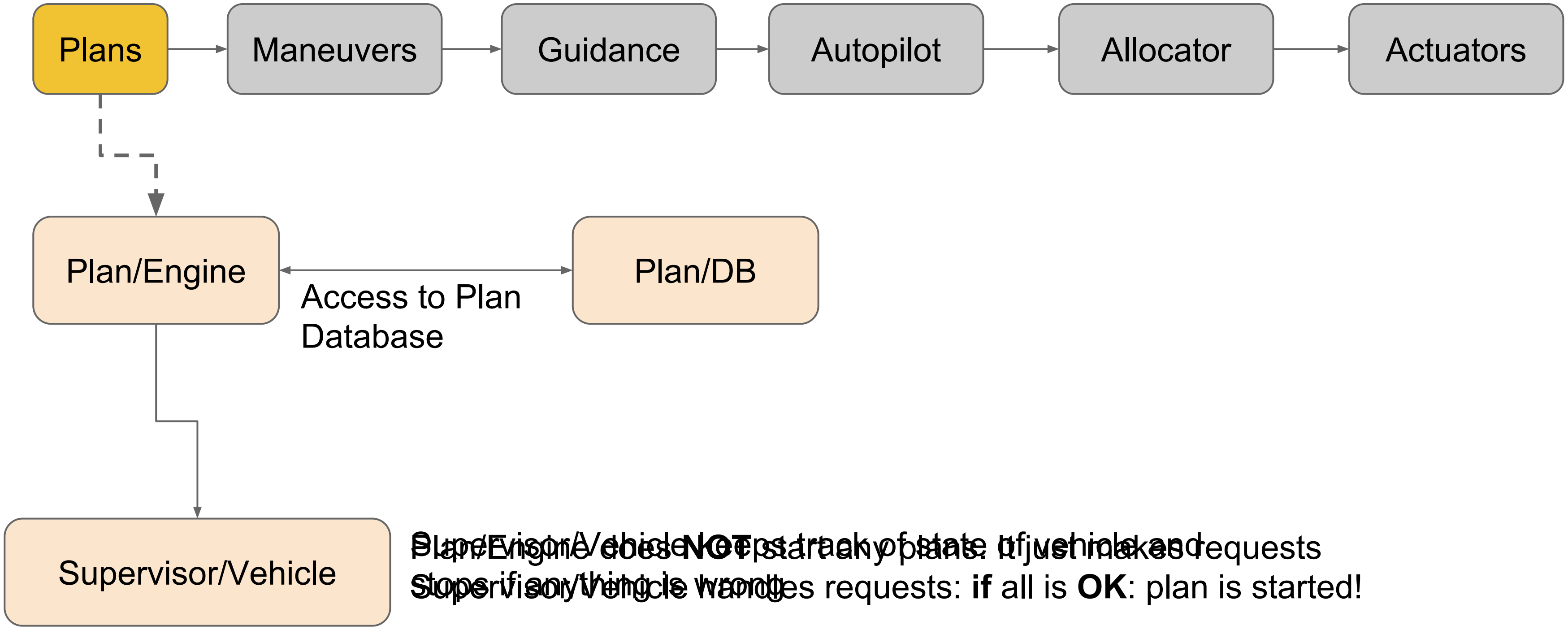
IMC message anatomy

Sync Num | Msg ID | Msg Size | Timestamp | Src Addr | Src Entity | Dst Addr | Dst Entity | Msg | CRC16

Control Architecture



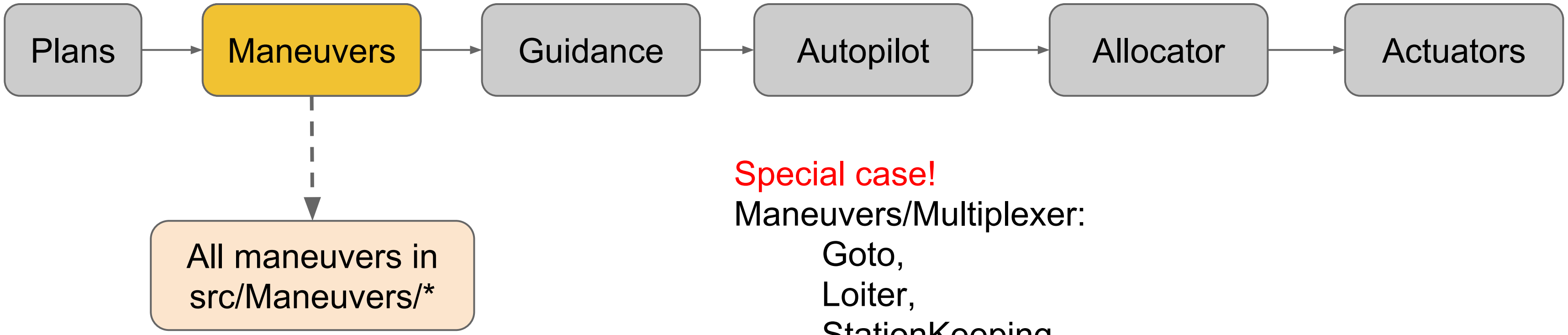
Control Architecture: LAUV



Supervisor/Vehicle keeps track of state of vehicle and stops if anything is wrong

Plan/Engine does **NOT** start any plans: it just makes requests
Supervisor/Vehicle handles requests: **if** all is **OK**: plan is started!

Control Architecture: LAUV



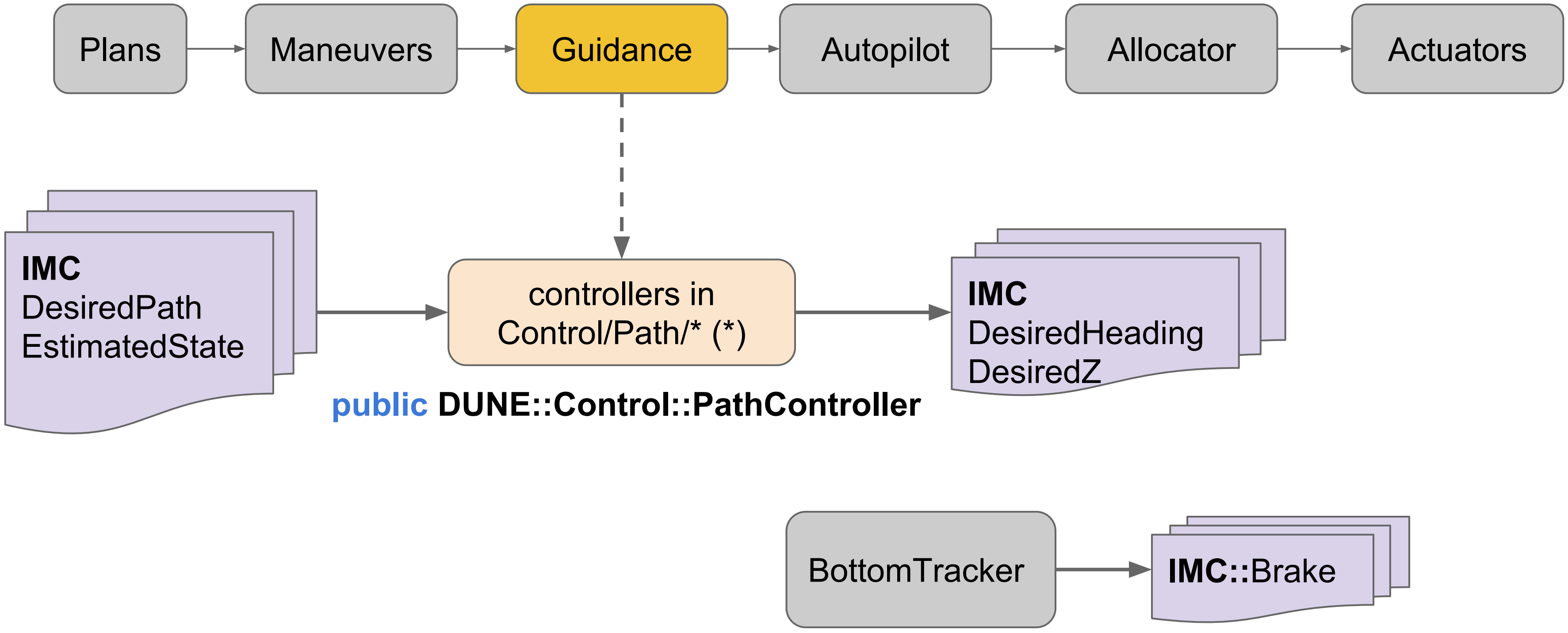
public DUNE::Maneuvers::Maneuver

Special case!

Maneuvers/Multiplexer:

- Goto,
- Loiter,
- StationKeeping,
- PopUp,
- Rows,
- Elevator,
- Dislodge,
- FollowPath,
- Launch

Control Architecture: LAUV



Plans

Maneuvers

Guidance

Autopilot

Allocator

Actuators

IMC
DesiredPath
EstimatedState

controllers in
Control/Path/* (*)

public DUNE::Control::PathController

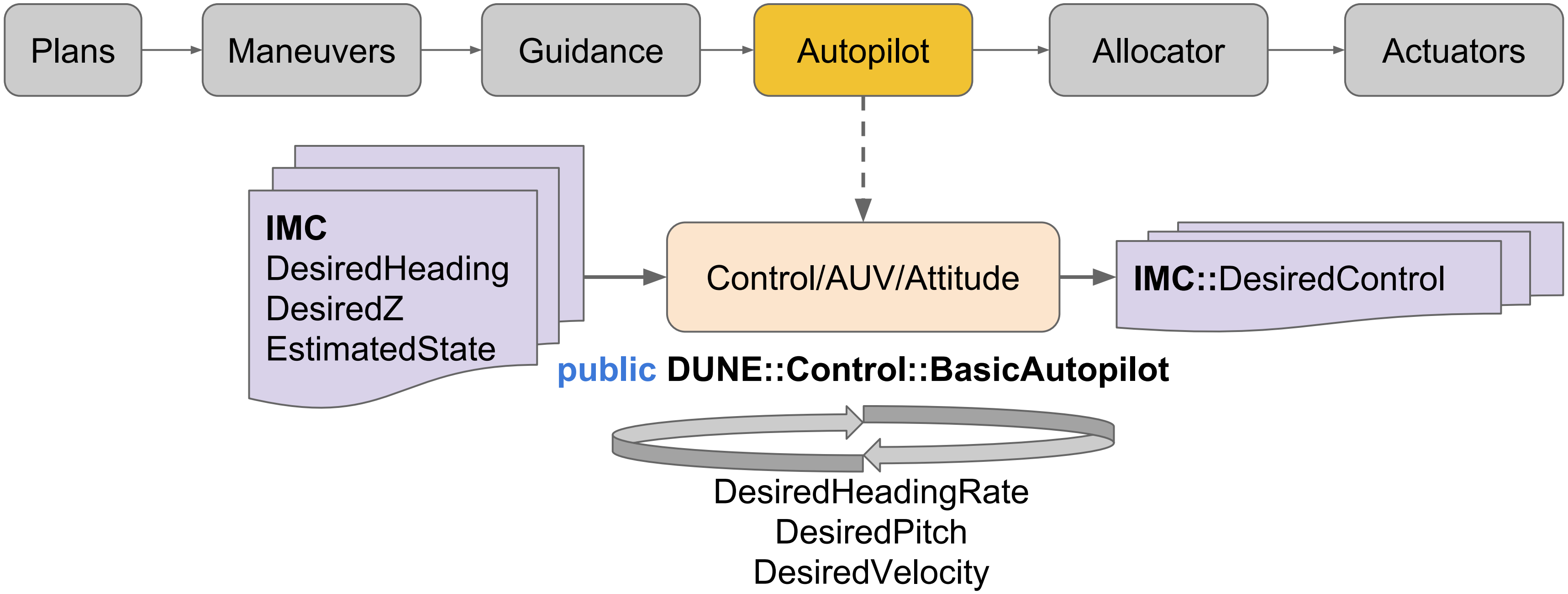
IMC
DesiredHeading
DesiredZ

BottomTracker

IMC:: Brake

(*) e.g: VectorField, ILOS, PurePursuit

Control Architecture: LAUV



Plans → Maneuvers → Guidance → Autopilot → Allocator → Actuators

IMC
DesiredHeading
DesiredZ
EstimatedState

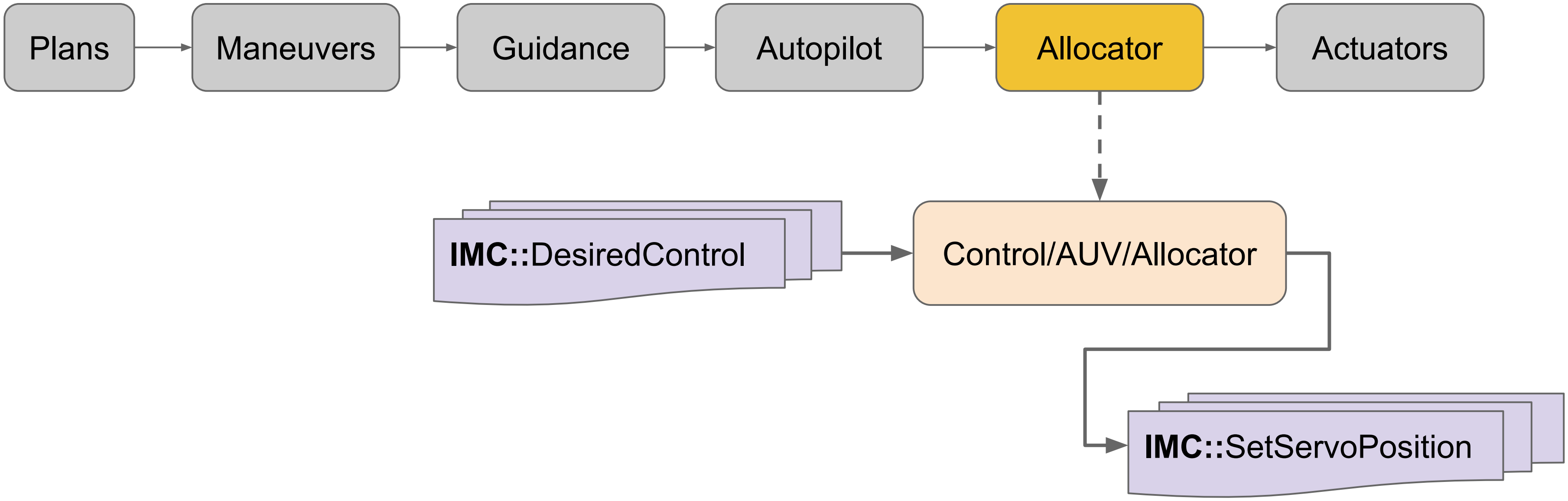
Control/AUV/Attitude

IMC:: DesiredControl

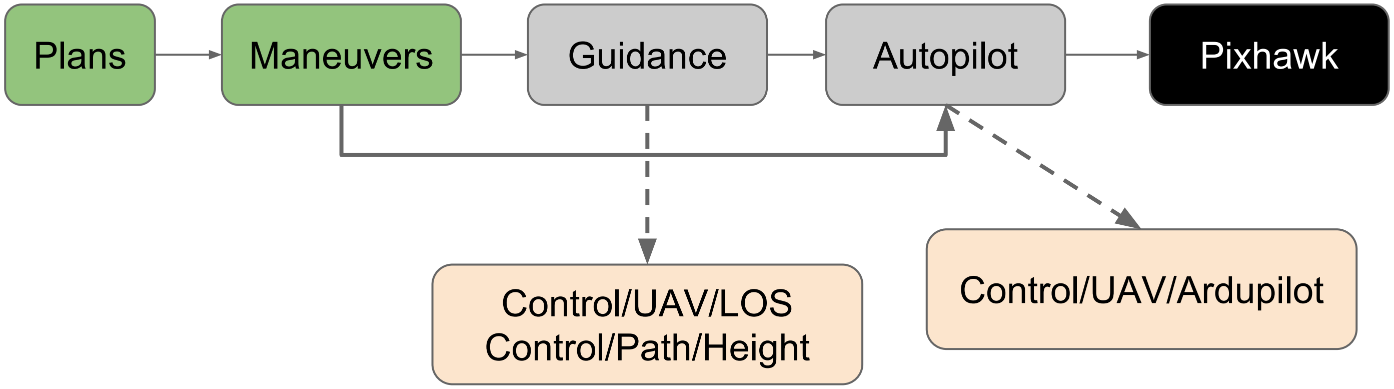
public **DUNE::Control::BasicAutopilot**

DesiredHeadingRate
DesiredPitch
DesiredVelocity

Control Architecture: LAUV



Control Architecture: UAV



Simulation & Replay

Simulate vehicle kinematic and sensors measurements:

```
./dune -c lauv-xpto -p Simulation
```

SIMULATION

Replay log of performed mission for navigation purpose:

```
./dune -c testing/replays/sgnav-replay      // Starts replay, waits for IMC logged messages
./dune-sendmsg <ip> <port> ReplayControl 0 <path_to_log>   // Send IMC logged messages
```

In **dune/etc/testing/replays** you may find more replays or even create your replay config file.

REPLAY

DUNE core - Class database

Need something ?
Should that something **exist** already ?

1) Search

grep -ri “matrix” <path_to_dune_src>

- ./Maneuver/CoverArea/Task.cpp: **Math**::**Matrix** m_rows; // etc

2) Ask

   a) jbraga@lsts.pt; trodrigues@lsts.pt
   b) dune@lsts.pt
   c) rasm@oceanscan-mst
   d) lsts-toolchain@googlegroups.com

3) if it does not exist - **implement**

DUNE core - Class database

Hardware - **Serial Port**, GPIO, I2C, **UCTK**

Coordinates - Transformation between referentials, **WGS84**, UTM

Database - Connect, Run Statement, etc

IMC - To deal with IMC messages (parser, serialization, json)

Math - You cand find almost every math functions, **matrix** operations, derivative, etc

Network - TCP, **UDP**, TDMA, etc

Parsers - **NMEAReader/Write**, PlanConfigurations, etc

Time - Delay, Delta, **Counter**, etc

Utils - **String**, XML, NMEA parser, **ByteCopy** (big/little endian), etc

Periodic tasks

- **Periodic** class inherits from **Task** class
  - **class** Periodic: **public** Task

- *onMain(void)* calls virtual *task(void)* at a fixed (configurable) frequency.
  - Tasks can inherit from **Periodic** (instead of class **Task**) - the body where implementation goes is *task(void)* instead of *onMain(void)*

- “Execution Frequency” is the argument that changes task frequency (default: 1 Hz)

EntityState

Each Task has an associated entity state.

EntityState is the state of the task, that can be seen from HTTP server (<ip>:8080)

Possible entity states:

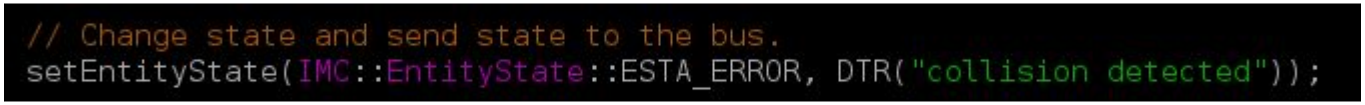
- BOOT
- NORMAL
- FAULT
- ERROR
- FAILURE

When DUNE boots, all tasks are at BOOT state. Depending on implementation and needs, the entity state should be updated.

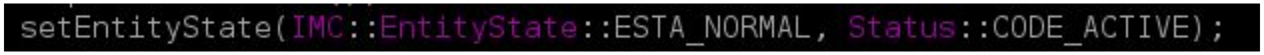
The most commonly used states are NORMAL and ERROR

EntityState

- To change state use *setEntityState*

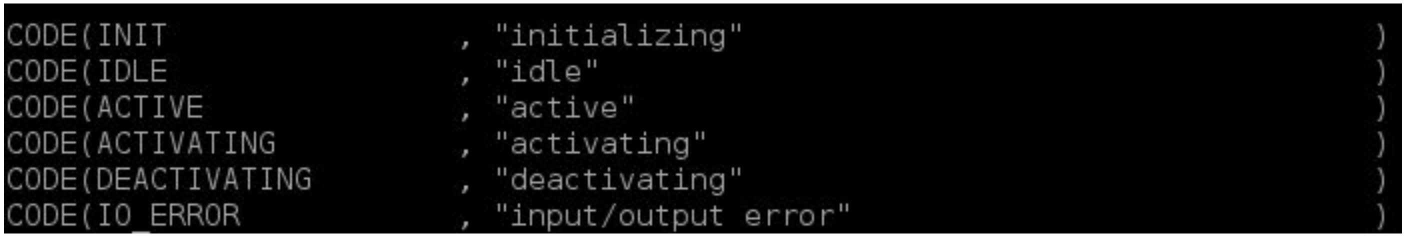


```
// Change state and send state to the bus.
setEntityState(IMC::EntityState::ESTA_ERROR, DTR("collision detected"));
```



```
setEntityState(IMC::EntityState::ESTA_NORMAL, Status::CODE_ACTIVE);
```

- *Status* is a class that translates codes into commonly used Strings.
    - Please check all codes in *src/DUNE/Status/Codes.def*



```
CODE(INIT                , "initializing"                                   )
CODE(IDLE                , "idle"                                           )
CODE(ACTIVE              , "active"                                         )
CODE(ACTIVATING          , "activating"                                     )
CODE(DEACTIVATING        , "deactivating"                                   )
CODE(IO_ERROR            , "input/output error"                             )
```

Output messages

- Do **not** use **std**::cout(), printfs() etc
- Tasks stream functions should be used
  - They guarantee messages are logged, and
  - written to the **Output.txt** file

- inf() // information
- war() // warnings
- err() // error messages

All these messages should implement DTR macro
e.g: inf(DTR("running again"));

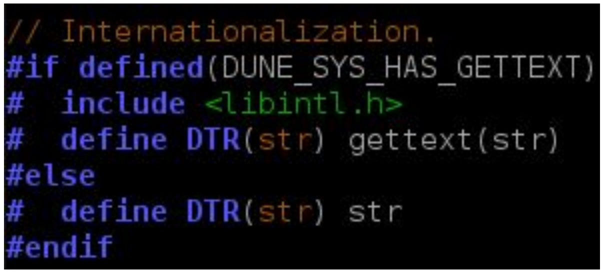
Debug (developer oriented) messages

- debug()
- trace()
- spew()

Debug Level = None	// no messages are sent
Debug Level = Debug	// only debug is sent
Debug Level = Trace	// debug + trace
Debug Level = Spew	// **all** debug goes

What is DTR macro ? I18N

in DUNE/Config.hpp.in:



```
// Internationalization.
#if defined(DUNE_SYS_HAS_GETTEXT)
#  include <libintl.h>
#  define DTR(str) gettext(str)
#else
#  define DTR(str) str
#endif
```

It’s used to mark strings for internationalization

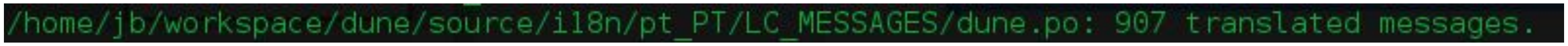
Folder **<dune_source>/i18n** has the implemented translations.

Fix translations:

in DUNE build folder run (check cmake/I18N.cmake for details):

1. make i18n_extract
2. make i18n_update
3. make i18n_compile

pt_PT translation outcome:



```
/home/jb/workspace/dune/source/i18n/pt_PT/LC_MESSAGES/dune.po: 907 translated messages.
```

To fix: edit *<path_to_dune>/i18n/pt_PT/LC_MESSAGES/dune.po* file and rerun commands. Then rerun i18n commands to validate and commit.

Doxygen: generating documentation

DUNE uses doxygen tags for documentation.
Each tag starts with ‘**//!**’

- @param[in] *name* <description>   // input parameter
- @param[out] *name* <description>   // output parameter
- @return <description>   // function return

Please check:
src/Vision/DFK51BG02H/Task.cpp ‘//!’ tags and respective documentation.

Use **//!** to introduce member variables and describe task methods.
Everything else use ‘**//**’

Git - How to use

Git is used for software version control.

Good Practices

- Read: *DUNE Git Manual* (available on Drive)
- Read: Github wiki - Git: **Introduction** / **Commit Messages** / Releases
- Respect **ALL** of the rules above. You will adapt to us, not the other way around.
- Never commit compiled files
- Commit only files you have changed
- Do not commit files that will make dune uncompilable
- Never keep files checked out for too long
- Always update your working copy before start working
- When merging, do not fast forward
- Atomic commits

Style Guide

1. Respect the style guide.
2. Respect the style guide.
3. Respect the f**** style guide.

….

Seriously, **follow the style guide** but also look at other tasks' programming and try to follow it. It helps a lot when a project has several contributors if the style is somewhat uniform - then it's really easy to jump in, analyze and fix/add something.

Also, do not write **everything** into a single Task.cpp file. Identify and divide by self-contained, well documented classes with clean and easy APIs.
e.g: Transports/Evologics, Transports/SUNSET, Sensors/Imagenex837B

The end.. relax

if you’ve reached the end of this presentation you’re either desperate or a fool. Here’s the “easter egg”

Programming is a lot like sex. One mistake and you're providing support for a lifetime.