**Qibuild:**

* is a generic framework that helps for managing several projects and their dependencies
* makes compilation easy
* by default uses libraries provided by your system (toolchain)
* is truly cross - platform
* based on Cmake
* composed of 2 parts: qibuild cmake framework (CmakeLists.txt) and the qibuild/qitoolchain command line tools, that helps build projects while taking dependencies into account and generate re-distributable binary packages
* to create a work tree - “qibuild init”
* Qitoolchain create mytoolchain /path/to/cpp/SDK/toolchain.xml --default
* create new project:
  + qisrc create myproject
    - Main.cpp
    - Test.cpp
    - CMakeLists.txt
    - Qproject.xml (must be represent for qiBuild to know how to build the project)
  + qibuild configure myproject ( qibuild configure --release myproject)
  + qibuild make myproject (qibuild make --release myproject)

**Naoqi** **framework**:

* Naoqi is the name of the main software that runs on the robot and controls it
* Naoqi framework is the programming framework used to program NAO
* Cross-platform
* Cross-language
* Introspectation = the framework knows which functions are available in the different modules where

**C++ SDK:**

* is used to develop with IDE
* to use the naoqi framework on your PC
* on Linux you can cross-compile libraries to embed in nao
* there are a few things that are c++ specific, the key difference - 2 types of proxies
  + Specialized proxies (they correspond to Aldebaran features like ALMotion, ALTextToSpeech etc. ; they give direct access to already existing methods + they give you compile-time type checking).

#include <alproxies/altexttospeechproxy.h>

const std::string phraseToSay = "Hello world";

AL::ALTextToSpeechProxy tts("nao.local", 9559);

tts.say("Hello world");

* + - * Generic proxies (they give access to any module including the ones which also have specialized proxies; have no information about the methods which are bound to these modules 🡪 the user must specify himself the name and parameters of the methods: if there is a mistake 🡪 exception; is slower and more error-prone, but very flexible (can adapt to any module) ).

#include <all common/alproxy.h>

const std::string phraseToSay =" hello world";

AL::ALProxy proxy("ALTextToSpeech", " nao.local", 9559);

Proxy.callVoid("say", phraseToSay);

// or if the method returns smth

bool ping = proxy.call<bool>("ping");

**Creating a new module:**

* + - * qibuild create mymodule (qisrc create mymodule)
      * Structure of a module:
      * CMakeLists.txt (is a script file that will be read by CMake to generate makefiles, or Visual Studio solutions)
      * Main.cpp
      * Qibuild.cmake (must be included by the CMakeLists.txt to find the qiBuild CMake framework)
      * Qibuild.manifest (must be present for qiBuild to know how to build the project)
  + Standart CMakeLists.txt

cmake\_minimum\_required(VERSION 2.8)

#give a name to the project

project(mymodule)

# you need this to find the qiBuild Cmake framework

include("qibuild.cmake")

#create a executable named mymodule

#with the source file main.cpp

qi\_create\_bin(mymodule "main.cpp")

#to import libraries

qi\_use\_lib(projectname library1 library2)

* + - also you can create a remote module
    - to update CmakeLists.txt ( to add new source files)
      * set( variableName sourceFile1 sourceFile2)
      * qi\_create\_bin(mybroker ${variableName}
    - to update main.cpp
      * create a broker .........
  + to start remote module
    - ./mymodule --pip robotIP -pport robotPort
    - or to change autoload.ini
  + to create library ( qi\_create\_lib(for boost::shared\_ptr) вместо q\_create\_bin )
  + to create local module (plugin) you need entry and exit points, no main function
    - int\_createModule(boost::shared\_ptr<AL::ALBroker> broker)
    - int\_closeModule()
  + to start local module
    - Autoload.ini
    - dynamic linking loader ( to use it in another module)
  + it is possible to switch from local to remote

**C++ library API:**

* Libqi - standard portable functions, include utilities for managing (access to OS and system level functionalities)
  + application path access( data, conf ..)
  + logging
  + SO abstraction
* Libalerror - contains AL::ALError and AL::ALNetworkError
* Libalvalue - contains AL::ALValue (generic container, can be int, bool, float, string, array)
  + #include <alvalue/alvalue.h>
  + AL::ALValue val;
  + val.arrayPush(1);
  + val.arrayPush("hello");
  + And other ways
  + Difficult to store map
  + Double is not supported
  + Not all convectors from std::vector to AL::ALValue
* Libalcommon - to create naoqi modules and communicate between them (broker, module, proxy, signature, main function)
* Libalextractor - to receive hardware data (sensors, audio, video)
* Libalmemoryfastaccess - fast read/write access to some atomic variables of ALMemory (float/int)
* Libalmath - optimized mathematic toolbox for robotics, give access to main types and tools of kinematics and dynamics, all the arguments and return values are given in SI units: meters, radians..
* Libalaudio - an audio extractor to receive audio buffer
* Libalvision - a video extractor to receive video buffer

**Debugging C/C++ on the robot:**

* 2 ways:
  + locally (running gdb on robot)
  + remotely ( running gdbserver on the robot and gdb on the host computer)
* cross-compiled binaries - only locally
* locally
  + on the robot, just run
    - gdb naoqi-bin
  + then execute any gdb commands you want
* remotely
  + only on the Linux host
* using gdb
  + can manage breakpoints
  + execution inside gdb
    - start
      * (Gdb) run <arg1> <arg2>
    - to pause
      * Ctrl-C
    - to quit
      * (Gdb) quit
    - when a breakpoint is triggered
      * display the source code around the breakpoint
        + (gdb) list
      * display the backtrace
        + (gdb) bt
      * move up and down in the backtrace
        + (gdb) up
        + (gdb) down

**Naoqi modules APIs:**

* Core modules - are always available. Every module comes with a list of default methods
  + ALBehaviorManager - start and stop behaviors
  + ALConnectionsManager - manage connection to a network and it's configuration
  + ALMemory - get and insert data for every other module to use
  + ALModule - create your own modules
  + ALPreferences - read and save settings from configuration files
  + ALResourceManager - handle resources
  + Deprecated modules
    - ALBonjour
    - ALLauncher - load dynamic libraries inside NAOqi and runs executables
    - ALLogger - logs information on the robot from a remote machine
* Motion module - provides methods which facilitate making NAO move. 4 major groups of methods for controlling the:
  + Joint stiffness (basically motor on/ off)
  + Joint position
  + Walk
  + Robot effector in the Cartesian space (whole body constraints etc)

+ implement some reflexes such as self-collision avoidance, fall manager and smart stiffness

* Audio
  + Sound management
    - ALAudioDevice - audio inputs and outputs
    - ALAudioPlayer
    - ALAudioRecorder
  + Sound detection and localization
    - ALSoundDetection
    - ALAudioSourceLocalization
  + + you can add languages
* Vision
  + ALBacklightingDetection
  + ALDarknessDetection
  + ALFaceDetection
  + ALLandmarkDetection
  + ALMovementDetection
  + ALPhotoCapture
  + ALRedBallCapture - to detect red and circular objects
  + ALVideoDevice
  + ALVideoRecorder
  + ALVisualRecognition
  + ALVisualCompass
  + ALVisualToolbox (deprecated ) - provides tools to record / analyse camera images
* Sensors
  + High level
    - ALSensors
    - ALBattery
    - ALFse - generates the event footContactChanged()
    - ALInfrared
    - ALLaser
    - ALSonar
    - ALRobotPose (deprecated), use ALRobotPosture instead
  + Low level
    - ALLeds
* Trackers - allow you to make NAO track targets (a red ball or a face)
  + ALFaceTracker
  + ALRedBallTrackee
* DCM - is a software module, part of the naoqi system, that is in charge of the communication with all electronic devices in the robot (boards, sensors, etc) except the camera and the sound

**Types naoqi:**

* standard - void, bool, int, float, std::string, std::vector<std::string>, std::vector<float>
* specific
* ALValue
* boost::shared\_ptr<AL::ALOwner>
* boost::shared\_ptr<AL::ALHierarchyOwner>
* AL::ALImage - generic image container

**NAOFlasher:**

* is a tool allowing you to update your NAO Operating system (OpenNAO)

**The NAOqi process:**

* The NAOqi executable which runs on the robot is a broker. When it starts, it loads a preferences file called *autoload.ini* that defines which libraries it should load. Each library contains one or more modules that use the broker to advertise their methods.

**Broker:**

* a broker is an object that provides two main roles:
  + it provides directory services: Allowing you to find modules and methods.
  + it provides network access: Allowing the methods of attached modules to be called from outside the process.

**Proxy:**

* + a proxy is an object that will behave as the module it represents. For instance, if you create a proxy to the ALMotion module, you will get an object containing all the ALMotion methods.
  + to create a proxy to a module, (and thus calling the methods of a module) you have two choices:
    - simply use the name of the module. In this case, the code you are running and the module to which you want to connect to must be in the*same* broker. This is called a *local* call.
    - use the name of the module, and the IP and port of a broker. In this case, the module must be in the corresponding broker

**Modules:**

* + can be either remote or local.
    - If it is remote, it is compiled as an executable file, and can be run outside the robot. Remote modules are easier to use and can be debugged easily from the outside, but are less efficient in terms of speed and memory usage.
    - If it is local, it is compiled as a library, and can only be used on the robot. However, they are more efficient than a remote module.
  + Local modules are two (or more) modules launched in the same process. They speak to each other using only **ONE** broker
  + Remote modules are modules which communicate using the network. A remote module needs a broker to speak to other modules
    - A connection Broker to Broker opens a mutual communication. Modules from both brokers can talk to each others.

For example, you have two modules B and C. When you connect their brokers, B can access to C’s functions and C can access to B’s functions.

To connect modules this way you need to specify the IP address and port number of the main broker. (--pip, --pport command line option when you start your module). Then you can access the module by getting a proxy on it:

AL::ALProxy proxy = AL::ALProxy(<modulename>);

Since module’s broker is already connected using --pip and --pport, you do not need to specify IP address and port number when you create a proxy.

* + - A Proxy to Broker connection opens a single way of communication. The proxy can access to all modules registered to the broker BUT the modules registered to the broker cannot access to the module that owns the proxy.

You can connect your module to another one without specifying --pip and --pport. To do that, you need to create a proxy inside your module and connect it to the broker IP address and port number you want.

For example, you have two modules B and C. When you connect B to C just using a proxy, B can access to C functions BUT C cannot access to B functions.

AL::ALProxy proxy = AL::ALProxy(<modulename>, <ip\_adress>, <port\_number>);