FP6-004381-MACS

MACS

Multi-sensory Autonomous Cognitive Systems Interacting with Dynamic Environments for Perceiving and Using Affordances

Instrument: Specifically Targeted Research Project (STReP)

Thematic Priority: 2.3.2.4 Cognitive Systems

D1.1.2 Specification of Module Interfaces

Due date of deliverable: May 31, 2005
Actual submission date v3: July 16, 2007

Start date of project: September 1, 2004 Duration: 39 months

Fraunhofer Institute for Intelligent Analysis and Information Systems (FhG/AIS)

Revision: Version 3

Project co-funded by the European Commission within the Sixth Framework Programme (2002–2006)		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

EU Project MACS

Deliverable 1.1.2

Specification of Module Interfaces

Rainer Worst, Claus Hoffmann, Björn Wingman, Maya Cakmak, Martin Hülse

Number: **MACS/1/1.2**
WP: *1.1*
Status: *Version 3*
Created at: *February 8, 2005*
Revised at: *May 29, 2007*

FhG/AIS	Fraunhofer Institut für Autonome Intelligente Systeme, Sankt Augustin, D
JR_DIB	Joanneum Research Graz, A
LiU-IDA	Linköpings Universitet, Linköping, S
METU-KOVAN	Middle East Technical University, Ankara, T
OFAI	Österreichische Studiengesellschaft für Kybernetik, Vienna, A

This research was partly funded by the European Commission's 6th Framework Programme IST Project MACS under contract/grant number FP6-004381. The Commission's support is gratefully acknowledged.

Author addresses:

Rainer Worst
Fraunhofer Institut für
Autonome Intelligente Systeme
Schloß Birlinghoven
D-53754 Sankt Augustin, Germany

Fraunhofer Institut für
Autonome Intelligente Systeme
Schloss Birlinghoven
D-53754 Sankt Augustin
Germany

Tel.: +49 (0) 2241 14-2683
(Co-ordinator)

Contact:
Dr.-Ing. Erich Rome

Joanneum Research
Institute of Digital Image Processing
Computational Perception (CAPE)
Steyrergasse 9
A-8010 Graz
Austria

Tel.: +43 (0) 316 876-1769

Contact:
Dr. Lucas Paletta

Linköpings Universitet
Dept. of Computer and Info. Science
Linköping 581 83
Sweden

Tel.: +46 13 24 26 28

Contact:
Prof. Dr. Patrick Doherty

Middle East Technical University
Dept. of Computer Engineering
Inonu Bulvari
TR-06531 Ankara
Turkey

Tel.: +90 312 210 5539

Contact:
Prof. Dr. Erol Şahin

Österreichische Studiengesellschaft
für Kybernetik (ÖSGK)
Freyung 6
A-1010 Vienna
Austria

Tel.: +43 1 5336112 0

Contact:
Prof. Dr. Georg Dorffner

Contents

1 Introduction

In this document we describe the methods to connect MACS components, that are written in different programming languages like C, C++ or Java. Our goal is to enable the developers in WP2 – WP5 to implement their modules and interfaces according to these standards.

You find a description of all steps to download and install the necessary middleware for this purpose. Furthermore, a generic example is described, that demonstrates how the specified mechanism works.

We will use CORBA as the glue between the different software components that establish the MACS architecture. CORBA is the acronym for Common Object Request Broker Architecture, OMG's open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. CORBA has been specified by the Object Management Group (OMG) [1].

Among the different implementations of CORBA that are available we have chosen TAO [2], which is an open-source CORBA-compliant Object Request Broker born of research at Washington University.

But the main part of this document is the detailed introduction and discussion of the interface specification of the KURT3D system. This specification is the base for the "synchronization" between FhG/AIS and METU-KOVAN in order to provide the same API for the KURT3D system in simulation (MACSim) and real.

2 Installation and using TAO

To install TAO, look at the instructions on [2]. They have recently included support for GNU autotools, but this feature seems not yet to work reliably. The traditional method is described below. More detailed information may be found on:

`http://www.cs.wustl.edu/~schmidt/ACE_wrappers/ACE-INSTALL.html`
and on:
`http://www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO/TAO-INSTALL.html`.

You will need the version 1.4.3, which is the one that is used in MACS. Download the TAO distribution from:

`http://deuce.doc.wustl.edu/ACE+TAO.tar.bz2`

You can also find a copy of the TAO distribution on the CVS server in the directory `macs/tools`. Unpack this in the location you intend it to be installed. There is no `'make install'` step for this distribution. `/usr/local` is a common place, so the TAO package will unpack into `/usr/local/ACE_wrappers`.

Optionally rename `ACE_wrappers` to `ACE_wrappers-1.4.3` according to the version you are using.

Set two environment variables, `ACE_ROOT` and `TAO_ROOT`. If you unpacked TAO into `/usr/local/ACE_wrappers-1.4.3`, use these values:

```
setenv ACE_ROOT /usr/local/ACE_wrappers-1.4.3
setenv TAO_ROOT ${ACE_ROOT}/TAO
```

Configure for Linux:

```
cp ${ACE_ROOT}/ace/config-linux.h ${ACE_ROOT}/ace/config.h

cp ${ACE_ROOT}/include/makeinclude/platform_linux.GNU
   ${ACE_ROOT}/include/makeinclude/platform_macros.GNU
```

Then add `${ACE_ROOT}/ace` and `${ACE_ROOT}/lib` to `LD_LIBRARY_PATH`:

```
setenv LD_LIBRARY_PATH ${ACE_ROOT}/ace:${ACE_ROOT}/lib:$LD_LIBRARY_PATH
```

Build:

```
cd ${ACE_ROOT}/ace/
make
```

Then build TAO:

```
cd ${ACE_ROOT}/apps/gperf
make
cd ${TAO_ROOT}
make
```

Warning: The execution of the last `'make'` may last some hours!
Finally, you have to set up your environment like this (e. g., by inserting in `.cshrc`):

```
setenv ACE_ROOT /usr/local/ACE_wrappers-1.4.3
setenv TAO_ROOT ${ACE_ROOT}/TAO
setenv PATH ${ACE_ROOT}/bin:${TAO_ROOT}/TAO_IDL:$PATH
setenv LD_LIBRARY_PATH ${ACE_ROOT}/lib:${LD_LIBRARY_PATH}
```

And so you may use CORBA TAO for the very first time generating the C++ template for an arbitary IDL file by executing:

```
tao_idl demoserver.idl
```

As you see you get nine new files:

```
demoserverC.cpp
demoserverC.h
demoserverC.inl

demoserverS.cpp
demoserverS.h
demoserverS.inl

demoserverS_T.cpp
demoserverS_T.h
demoserverS_T.inl
```

The next chapter give you an impression how you may organize your own CORBA environment and how to use CORBA to realize simple data exchange between two separate programs.

3 Three examples of using CORBA with C++ and Java

3.1 CORBA Demo by Björn Wingmann

You can find a demo provided by Björn Wingmann in the CVS of MACS (directory `macs/corbademo`). The programs `demo_server` and `demo_client` demonstrate how to use a CORBA server and client.

The server implements the IDL-interface `DemoServer`, which you can find in the file `idl/demoserver.idl`. This server keeps an internal map from strings to strings, and allows inspection and manipulation of the strings via this interface. The interface is actually implemented in `DemoServerImpl.cc`.

The client creates a simple shell for using the server; a session could look like this:

```
~> demo_client
     -ORBInitRef NameService=corbaloc:iiop:porter:20000/NameService
> lookup foo
Name not found: foo
> store foo ninetysix
> lookup foo
ninetysix
> quit
```

This example uses the `lookup` and `store` functions from the interface to store the string pair `foo:ninetysix` in the server.

COMPILING, short version:

1. Checkout `macs/corbademo` from the CVS server.
2. Run the `./bootstrap.sh` script to set up autoconf.
3. Run `./configure`, add any options you feel are necessary. (See `configure --help`, e.g. `--prefix=$HOME` to change the output directory for the target 'install')
4. Do `make`.
5. Do `make install`.

RUN things like this (assuming that you are using port 20000 on a host named porter and that `demo_server` and `demo_client` are located in your PATH):

```
$TAO_ROOT/orbsvcs/Naming_Service/Naming_Service
  -ORBEndPoint iiop://porter:20000
demo_server -ORBInitRef NameService=corbaloc:iiop:porter:20000/NameService
demo_client -ORBInitRef NameService=corbaloc:iiop:porter:20000/NameService
```

There are two more useful options for the execution of `demo_server` resp. `demo_client`, which may be used for debugging:

```
-ORBDebugLevel 10 -ORBDottedDecimalAddresses 1
```

The first option starts a verbose mode with a lot of messages that show what is going on. The second option allows you to use IP-Addresses (e. g., 192.168.1.35) instead of host names, which may be useful if there are problems with the DNS at your site.

3.2 The light version of CORBA Demo

As someone who just needs a brief demonstration of CORBA TAO your may satisfied with the example above. As a software developer you may ask what are the essential needs to realize your own software projects encapsulated by CORBA TAO.

The most confusing of the usage of CORBA TAO is the makefile structure. Therefore we introduce in the following a minimal example that should help to setup up your environment for compiling your own CORBA TAO based applications.

We use the same example `corbademo` as above. But now we have only the IDL-file defining the interface and the four C++ files: `server.cc`, `client.cc`, `DemoServerImpl.h`, `DemoServerImpl.cc` and (of course) a makefile; and all this in a separate folder (called `corba_demo_light` if you check it out from the CVS repository via the command `cvs co macs/corba_demo_light`). You already now the meaning of these files from the explanations above, for that reason we only introduce the makefile and its structure. The structure of the makefile is taken from the web site:

`http://www.crossleys.org/~jim/corba/tao.html`

There you can find more details.

Open the file and have a look. First, you have to define your target(s), i.e., your executables. In our case, we have the two applications `client` and `server`.

```
# Ultimate executable targets
BIN = server client
```

The needed object files are defined as follows:

```
# Object files required for server
SERVER_OBJS = \
        demoserverC.o \
        demoserverS.o \
        DemoServerImpl.o \
        server.o

# Object files required for client
CLIENT_OBJS = \
        demoserverC.o \
        demoserverS.o \
        client.o
```

Notice, the object files `demoserverC.o` and `demoserverS.o` belong to the so-called IDL "stubs and skeletons" and are automatically generated according to the IDL file. The IDL file itself is here only indirectly given by means of the variable `IDL_SRC` referring to the automatically generated files from the IDL file `demoserver.idl`.

```
# Establish SRC target
IDL_SRC = demoserverC.cpp demoserverS.cpp
PROG_SRCS = client.cc server.cc
SRC = $(IDL_SRC) $(PROG_SRCS)
```

According to your dependencies, you have to add your external libraries and location of header files.

```
ACE_LDFLAGS= -L$(ACE_ROOT)/ace -L$(ACE_ROOT)/lib -lACE -lpthread -lrt
LDFLAGS += -L$(TAO_ROOT)/orbsvcs/orbsvcs -L$(TAO_ROOT)/tao
CPPFLAGS += -I$(TAO_ROOT)/orbsvcs
```

Finally, the targets (executables) are created according to the following rules:

```
# How to build server
server: $(addprefix $(VDIR),$(SERVER_OBJS))
          $(LINK.cc) $(LDFLAGS) -o $@ $^ $(VLDLIBS) $(POSTLINK)

# How to build client
client: $(addprefix $(VDIR),$(CLIENT_OBJS))
          $(LINK.cc) $(LDFLAGS) -o $@ $^ $(VLDLIBS) $(POSTLINK)
```

In summary, we introduced this example in order to give you a minimal setup for your own environment and that you get a feeling what is needed using CORBA. Please try it by executing the the commands `make`, `make depend`, `make clean` and `make realclean` and see what happens!

3.3 Connecting a Java Demo Client to TAO

The CORBA technology is part of the Java2 platform. In the Java2 platform you can find a Java CORBA ORB and the Interface Definition Language (IDL). IDL specifies interfaces for distributed applications that are independent of specific programming languages. A mapping from IDL to several programming languages, like JAVA, C, C++, and others is defined by OMG.

To learn about using CORBA please visit the following tutorial, that walks you through a simple "Hello World" example:

`http://java.sun.com/j2se/1.4.2/docs/guide/idl/tutorial/GSIDL.html`

Take the IDL interface that you can find in the file `idl/demoserver.idl` in the CVS for the demo client . To map the IDL interface to Java use the Java IDL Compiler `'idlj'`. If J2SE is installed, you can find it in the bin directory. The compiling command is:

```
idlj -fall demoserver.idl
```

Each statement in the OMG IDL is translated by the compiler to a corresponding statement in the Java programming language. The necessary CORBA stubs, skeletons, and ties for the demo example will be generated. You can store these files in a generated folder called Macs.

Create a file with the name DemoClient.java or take the example from the CVS repository (checking out `macs/corbademo` project, as described above).

Then you must compile your client. The compiling command is:

```
javac DemoClient.java Macs/Idl/*.java
```

To start the demo client, enter:

```
java -classpath Macs/Idl:. DemoClient
     -ORBInitRef NameService=corbaloc:iiop:porter:20000/NameService
```

... if the name service is still running on port 20000 of the host porter.

4 Specification of the KURT3D Modules

In this chapter we introduce the IDL interface specifications of the KURT3D robot system. The interfaces have been developed by METU-KOVAN and FhG/AIS in order to define the standard functionality delivered by the real platform and to provide a physics based simulation of this functionality (compare Del. 1.2.1 MACSim: Physics-based Simulation of the KURT3D Robot Platform for Studying Affordances).

The interface of the whole KURT3D system simply unifies four separate interfaces, which describe the functionality of the four main components of this system. These components are the basic control, servo webcams, laser scanner with 3D functionality and the crane. Hence, the IDL file defines a module, called `Macs`, containing the four interfaces `IBasicControl`, `ICamera`, `ILaserScanner3D` and `ICrane`.

```
#ifndef KURT3D_IDL
#define KURT3D_IDL
...

module Macs
{
    interface IBasicControl {...}

    interface ICamera {...}

    interface ILaserScanner3D {...}

    interface ICrane {...}
};
#endif
```

All partners are able to check out this IDL file from the CVS repository on gibson. Details concerning the CVS are described in Del. 1.1.3 Implementation of the software development environment. The following sections give a detailed description of each interface.

4.1 Basic control interface

The basic control interface provides basically the functionality that drives the robot. But simple sensor data are also delivered, such as proximity data, provided by the eight infrared (IR) sensors, and collisions, via the six bumpers. Additionally, the tilt sensor delivers data that serve as measurement for the robot's angles of inclination. Finally, also the current

movement of robot can be monitored. This can be done by reading the current (relative) encoder values and determine the time needed for theses values to accumulate. The movement of the robot can be defined by setting the PWM control parameters of the two motors driving the left and the right wheel.

In order to keep the overhead of CORBA during the data transmissions as small as possible we organized all the sensor data in one struct. In such a way all relevant sensor data can be delivered by only one request. The corresponding definitions can be found in the interface `IBasicControl` of module `Macs`.

```
interface IBasicControl
{
    const float          MAX_ROBOT_SPEED = 1.0;          // not used
    const unsigned short NMB_DISTANCE_SENSORS = 8;
    const unsigned short NMB_BUMPERS          = 6;

    typedef float        DistanceArray[NMB_DISTANCE_SENSORS];
    typedef boolean        BumperArray[NMB_BUMPERS];
    typedef long Tics;
    typedef double TimeDiff;


    struct TiltSensors   { float tiltX; float tiltZ;};

    struct Sensordata {
            DistanceArray  distance;
            BumperArray    bumper;
            TiltSensors    tilt;
    };

    boolean setPwmSignals(in long nLeftPwm, in long nLeftDirection,
                          in long nRightPwm, in long nRightDirection);

    boolean readTics(out Tics nTicsLeft,out Tics nTicsRight,
                     out TimeDiff dTimeDiff);
    ...
};
```

As one can see, IR sensor and bumper data are summarized by bounded arrays (of type `DistanceArray` or `BumperArray`). The values of the distance sensors are integers and are in the interval between 0 and 500. They are zero, if no obstacle has less than 100 cm distance to the sensor. The smaller the distance between sensor and obstacle, the larger the sensor value. Notice, this mapping is nonlinear. Details about the mapping between distance and corresponding sensor value are described in Del. 1.2.1 MACSim: Physics-based Simulation of the KURT3D Robot Platform for Studying Affordances. The maximal value of the IR sensors is 500. Therefore, we choose the `unsigned short` as data type for these values. Figure 1(a) indicates, which index of the array represents which distance sensor on the Robot system.

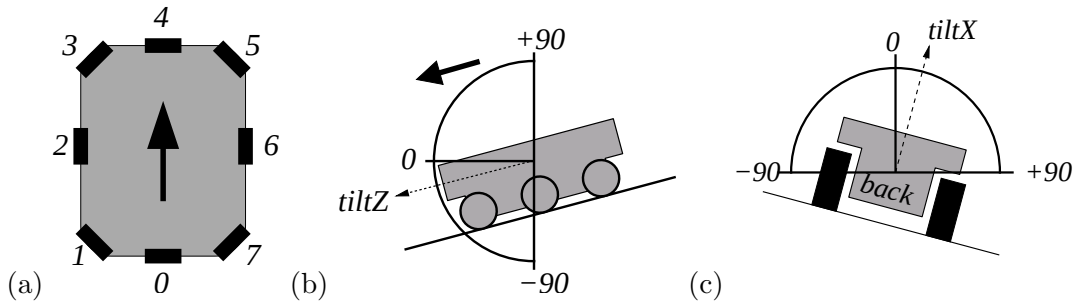


Figure 1: (a) Mapping between the index of the `DistanceArray` and actual distance sensor on the KURT3D platform. (b,c) Schema of the mapping between tilt sensor data and the inclination angles of the robot. The black arrow in (a) and (b) indicates the forward direction of the robot.

The tilt sensor data have two components `tiltX` and `tiltZ`. Each measures the angle of inclination. See Fig. 1(b) and (b) for details. The data type `float` is used to represent these values. The values are between -90 and 90, which represents the angles in degrees. Both values are summarized in the struct `TiltSensors`.

4.1.1 Configuration

First the function `configBasicControl(...)` has to be executed before other function calls can be executed.

The function

```
void robotMcReset();
```

realizes a reset of the whole basic control module. In fact, after this function call the robot stops moving, until the next commands (function calls) are executed.

4.1.2 Getter

For the sensor readings a function is defined, which delivers all data (organized in the struct `Sensordata`):

```
short getSensordata(out Sensordata data, out string msg);
```

But one is also able to request specific sensor data by the following functions:

```
short getDistanceData  (out DistanceArray distance);
short getBumperData    (out BumperArray   bumper);
short getTiltData      (out TiltSensors   tilt);
boolean readTics(out Tics nTicsLeft,out Tics nTicsRight,
                 out TimeDiff dTimeDiff);
```

Some of these functions have a return value of data type `short`. Such return values and the string `msg` of function `getSensordata` are intended for error codes and messages, in the case of problems. But these codes are not defined yet.

The method `boolean readTics(...)` reads the current (relative) encoder values and determines the time needed for theses values to accumulate. Notice:

- The encoder values are relative and are send every 10ms by the KURT robot.
- If (for timing reasons) more encoder messages have arrived, all of them will be read and the accumulated value will be returned.
- If there are no new encoder values, tic and time values of 0 will be returned.
- On the first call of this method the message buffer will be flushed and the result will always be 0.

If it is necessary to call this function, wait at least 10 ms before calling this function again, otherwise the CAN bus may be stuck!

- `nTicsLeft` left encoder value.
- `nTicsRight` right encoder value.
- `pldTimeDiff` time since the last call (i.e. the time in which these encoder values accumulated).
- `return value true` if encoder values are successfully read, out values are defined, `false` if out values are not defined.

4.1.3 Setter

```
boolean setPwmSignals(in long nLeftPwm, in long nLeftDirection,
                      in long nRightPwm, in long nRightDirection);
```

- `nLeftPwm` pwm value of the left wheels [0-1024]
- `nLeftDirection` direction of the left wheels [0,1]
- `nRightPwm` pwm value of the right wheels [0,1024]
- `nRightDirection` direction of the right wheels [0,1]
- `return value (boolean)` denoting if the CAN communication was successful

The method sets motor control parameters. The parameters are the pulse width modulation signal and a direction bit:

- PWM signal: Value between 0 and 1024 where 0 is maximum speed and 1024 is full stop.
- Direction bit: 0 for driving forward, 1 for driving backwards.
- Using this function the robot steers the wheels. I.e., if the robot is driving with a speed of let's say 50 cm/s and it is stopped by setting a PWM signal of 1024 to both wheels the robot will use its motors to stop immediately.

Important for the real KURT3D, after 100 ms without sending a pwm signals to the motor via this function call both motors are stopped. Hence, for our examples we call every 10 ms this function in order to keep the robot moving.

Further on, if it is necessary to call this function, wait at least 10 ms before calling this function again, otherwise the CAN bus may be stuck!

mode	vertical	horizontal
VGA	640	480
CIF	352	288
SIF	320	240
QCIF	176	144
QSIF	160	120
SQCIF	128	96

Table 1: Camera types and the corresponding resolution of the images

4.2 Camera interface

The camera interface actually defines the functionality of a pan-tilt camera. That means, one can independently control its horizontal and vertical orientation. Further on, the KURT3D system has two cameras of this type. Both are at the front on the left and right side.

Some cameras can be driven in different modes. This basically means, that the images will have different resolutions. We defined an enumeration type to handle the most common modes. But, in the MACS project all the partners are asked to use the VGA mode. See table 1 for the corresponding resolutions.

The raw image data are delivered by the camera as an array of chars. Each R/G/B pixel data of the raw image date is given by three chars, i.e. resolution of a pixel is 3x8 = 24 bit. Hence, each color value lies in range 0 and 255. The length of the array, which represents the whole image, is determined by the resolution of the camera. If the camera is running in VGA-mode 3x640x480 is the array length. The order of the color data per pixels is r,g,b and the pixels of the image are organized line-by-line.

For the transmission of these type of data via CORBA we choose the data type `octet` as basal type. Hence, the image data are send as a sequence of `octet`. This sequence is unbounded. The usage of `octet` avoids problems that occur when data containing specific codes that might be interpreted as termination signs. This is the case, if we use the data type `string`. As a consequence, incomplete data transmissions can occur. Using `octet` the image data are only handled as binary data and misguiding "interpretations" are avoided.

Further on, a second enumeration type is defined in order to configure a camera with respect to the side, which it is fixed. Such a configuration becomes necessary, because, as we will later see, a camera orientation to the left or the right will be based on equal function parameters.

The orientation of the camera is represented by integer values due to the underlying two servo motors. These two motors create the pan tilt orientation of the camera.

If both values, i.e. pan and tilt, are set to zero the camera is in the Null-position. Due to the assembly of the servo motors and the camera it is warranted that the Null-position is related to a "straight forward look" of the camera. That means, the optical axis of the camera is parallel to the "base axis" of the robot.

An increase of the pan position is related to an orientation to the right, a decrease means a orientation to the left. An increase of the tilt position is related to an upward orientation, a decrease means a downward orientation.

The terms left and right, up and down have to be understood from the robot's per-

spective. Hence, we have to distinguish between left and right camera. Otherwise, we get not the same mapping between parameter values (pan, tilt) and the resulting orientation for both cameras.

The qualitative relation between pan / tilt steps and the corresponding orientation of the camera is determined by the actual servo-motors. Notice, they even differ, if the same type of motor is used.

In the following, one can see the definition of the data types that provide the camera interface `ICamera`.

```
interface ICamera
{
    enum CamResolution{ VGA, CIF, SIF, QCIF, QSIF, SQCIF};
    typedef sequence<octet> RgbImageArray;
    enum Side {LEFT, RIGHT};
    typedef long Pan;
    typedef long Tilt;
    ...
};
```

4.2.1 Configuration

The following two functions are used to configure the camera and to get its current resolution.

```
short configCamera (in  Side          s,
                    in  CamResolution resolution,
                    in  Pan           panPos,
                    in  Tilt          tiltPos,
                    out string        msg);
```

All results of the other functions are undefined until this function is called. The return value and the string `msg` can be used for error codes and messages, but this issue is not defined yet.

4.2.2 Getter

The data that can be received from the cameras are obviously the resolution, the orientation, and, certainly, images. The two cameras of the KURT3D system can be distinguished by the side where they are fixed.

```
CamResolution getConfigCamera (in  Side s,
                               out Pan  panPos,
                               out Tilt tiltPos);

void    getCameraPosition (in  Side s,
                           out Pan  panPos,
                           out Tilt tiltPos);

boolean getRgbImage (in Side s, out RgbImageArray data);
```

The second function takes a snapshot with the configured resolution and returns the image data stored in a sequence of octet. If the return value is `FALSE` the image data are undefined.

4.2.3 Setter

The following functions are implemented to change the orientation of the cameras. This can be done by a direct setting of the servo motor positions (pan, tilt).

```
boolean setCameraPanTilt (in Side s,
                          in Pan panPos,
                          in Tilt tiltPos);
boolean setCameraPan  (in Side s, in Pan panPos);
boolean setCameraTilt (in Side s, in Tilt tiltPos);

boolean changeCameraPanTilt (in Side s,
                             in Pan panOffset, in Tilt tiltOffset);
boolean changeCameraPan  (in Side s, in Pan panOffset);
boolean changeCameraTilt (in Side s, in Tilt tiltOffset);
```

But the camera orientation can also incrementally be changed (see the last three functions). The values of `panOffset` and `tiltOffset` can be positive or negative. An increase of pan position represents an orientation to the right and vice versa. An increase of tilt position is related to an upward motion of the camera and vice versa.

Return value of all functions is `FALSE`, if the position could not be reached or if the offset exceeds the servo limits. Again, the servo limits are intrinsic properties of the camera systems and can not be defined here.

4.3 Laser scanner 3D interface

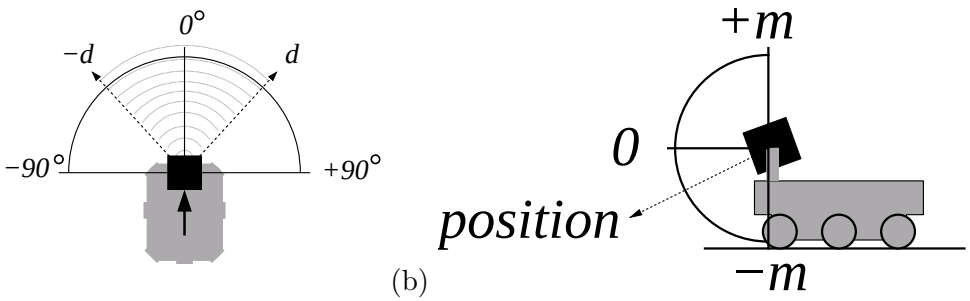
The laser scanner of the KURT3D system can run in two different modes: the 2-dimensional and the 3-dimensional mode. In both modes the laser scanner can deliver distance and remission data.

Starting with the 2-dimensional mode the following four parameters determine this type of scanning:

- `Resolution`,
- `ApexAngleHorizontal`,
- `Position`,
- `RemissionData`.

A laser scan can only have three different horizontal resolutions $\frac{1}{4}^\circ$, $\frac{1}{2}^\circ$ and 1°. Hence, an enumeration type represents these three resolutions.

The horizontal range of a 2D scan is symmetric and therefore the range is determined by a parameter `ApexAngleHorizontal` (see Fig. 2(a)). The angle value is an integer and lies in the range $0 \leq 90^\circ$. Given an apex angle of d and the resolution r, then a 2D scan



(a) (b)

Figure 2: Schema that indicates the definition of the parameters which determine a 2D scan. (a) horizontal range and (b) position of the scanner.

delivers $(d \cdot r) + 1$ data points, since angle zero has to taken into account. Hence, the maximal number of data points of one 2D scan is 721.

Additionally, one can determine the inclination of the laser scanner. Like the cameras, the tilting of the laser scanner is realized by a servo motor. The position zero of the servo motor is related to straight forward orientation of the camera (see Fig. 2(b)). An increase of the position represents up and a negative position a down of the scanner. The angle is determined by the used servo motor. The actual mapping between servo motor position and actual inclination angle is an intrinsic property of the servo motor. A common angle difference between two servo motor positions is 0.23°.

Finally, the parameter `EmissionData` determines, if emission data are required. If this value is `FALSE` no emission data will measured, which speeds up the scanning process of the scan on the real platform.

Considering the 3-dimensional scanning mode, obviously only the vertical component additionally has to be taken into account. Similar to the horizontal case, the vertical range is also symmetric and therefore determined by only one parameter `ApexAngleVertical`. But, now the values refer to relative servo motor positions according to the actual tilt position of the scanner. Further on, the vertically spanned region of a 3D scan might be scanned at different resolutions. In other words the servo can be stepped 1 by 1, 2 by 2 or n by n. The parameters of type `VecticalStep` will give the number of steps between two consecutive 2D scans in a 3D scan.

In the following one can see the part of the IDL specification, which defines the data types providing the laser scanner interfaces.

```
interface ILaserScanner
{
    enum Resolution {quarter, half, one};
    typedef boolean EmissionData;

    typedef long Position;
    typedef long ApexAngleHorizontal;
    typedef long ApexAngleVertical;
    typedef long VecticalStep;

    typedef short Dimension2D;
    struct Dimension3D {
        unsigned long vertical;
```

```
        unsigned long horizontal;
    };

    typedef sequence<double> Tscan2dDistanceArray;
    typedef sequence<double> Tscan2dEmissionArray;

    typedef sequence<Tscan2dDistanceArray> Tscan3dDistanceArray;
    typedef sequence<Tscan2dEmissionArray> Tscan3dEmissionArray;
    ...
};
```

The delivered 2D data are organized as an unbounded array of doubles. The dimension of the current 2D scan is stored in a variable of type `Dimension2D`. The 3D scan data are represented by an unbounded sequence of such 2D sequences. The dimensions of this 2 dimensional array are represented by a variable of type `Dimension3D`, which is a struct.

4.3.1 Configure

The data that a scan is delivering are only defined, if the corresponding configuration procedure is executed before scanning. The following two functions provide this functionality.

```
Dimension2D configLaserScanner2D(in  ApexAngleHorizontal apex,
                                 in  Resolution          res,
                                 in  Position            pos,
                                 in  EmissionData        emission,
                                 out string              msg);

Dimension3D configLaserScanner3D(in  ApexAngleHorizontal apexH,
                                 in  Resolution          resH,
                                 in  ApexAngleVertical   apexV,
                                 in  VecticalStep        verStep,
                                 in  Position            pos,
                                 in  EmissionData        emission,
                                 out string              msg);
```

The return values of both functions deliver the dimension of the data arrays which a scan is producing. *Notice, a new 3D configuration leads to a corresponding change of the 2D configuration and vice versa.*

If the values of the dimensions are negative, then the configuration failed. The string `msg` can be used to deliver same error codes or informations. But these codes are not defined yet.

4.3.2 Getter

Suppose the laser scanner is successfully configured, to get the desired data one has to start a scan process. The following two functions initiate and fulfill such a process.

```
boolean make2dScan(in Position pos);
boolean make3dScan();
```

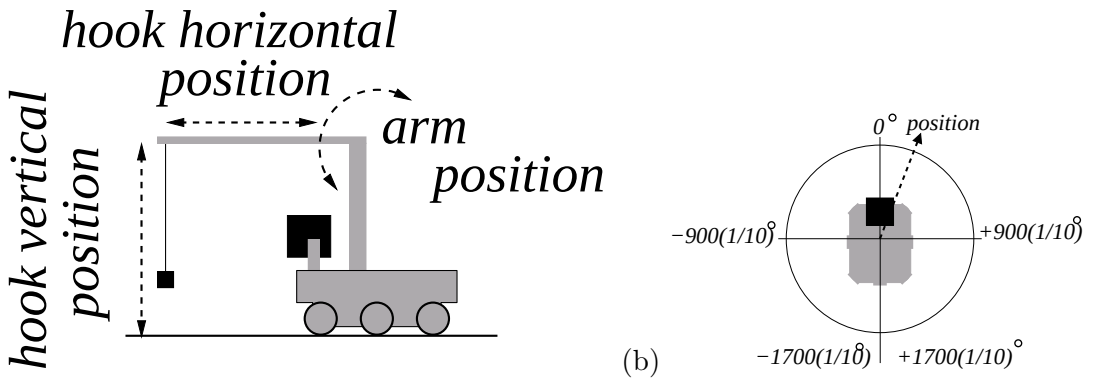


Figure 3: (a) The three degrees of freedom, which determines the crane movement. (b) Horizontal positioning of the crane and the corresponding ranges.

```
boolean is2dScanComplete();
boolean is3dScanComplete();
```

A scanning process might take a while. But as long as this process is not finished, no defined data can be delivered by the scanner. Therefore, functions are defined to get information about the scanning state. If the return value of function `is2dScanComplete()` or `is2dScanComplete()` is `FALSE` the scanning is not finished yet and the data are not defined.

If the return value is `TRUE` one can request the data with the following function calls.

```
boolean get2dDistance(out Tscan2dDistanceArray scan2dDistanceArray);
boolean get2dEmission(out Tscan2dDistanceArray scan2dDistanceArray);
boolean get3dDistance(out Tscan3dDistanceArray scan3dDistanceArray);
boolean get3dEmission(out Tscan3dEmissionArray scan3dEmissionArray);
```

If the return values are `FALSE` then the delivered data are undefined. This also can happen, if emission data are requested, but the scanner was configured without emission data.

Finally, the last two functions enable us to set and get the scanner position, without calling the configuration functions.

```
boolean setScannerPosition(in Position pos);
boolean getScannerPosition(out Position pos);
```

4.4 Crane interface

The crane has basically three degrees of freedom: arm position, vertical and horizontal hook position. Certainly, the magnet state and the speed of the motors are also free parameters, but for the discussion of the crane movement we stay focused on these three components.

The arm position (see Fig. 3 (a,b)) is given in degrees and its values lie in the interval $(-170°, +170°)$, i.e. the values are integer and the resolution is $1/10$ degree. The horizontal and vertical positions of the hook are given by positive integer values. These values are

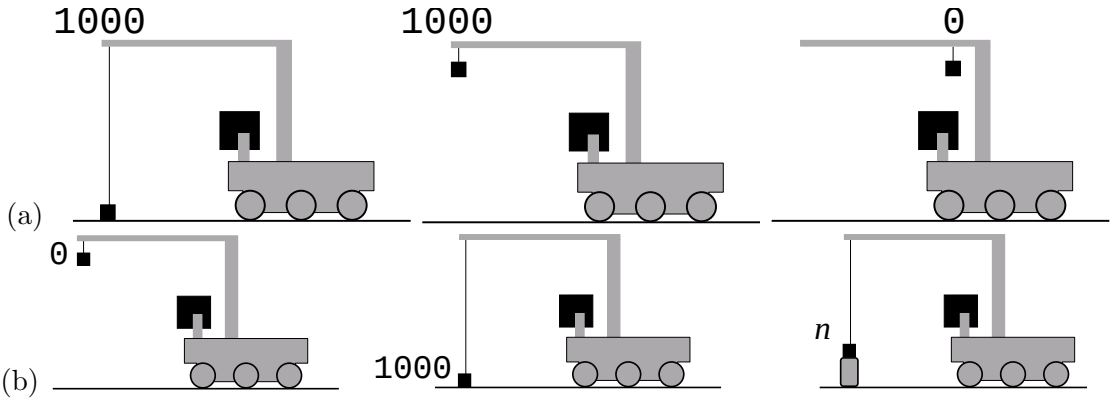


Figure 4: Mapping between horizontal (a) and vertical (b) hook position value and actual location of the hook.

between 0 and 1000. In Fig. 4 the mapping between position values and actual positions of the hook is indicated. This mapping is linear. Certainly, the implementation of this mapping needs a calibration process on the real robot. We explain this process later in this section in detail.

The null-position of the crane refers to a state that is defined as the magnet is switched off (i.e. nothing is carried by the crane), the arm is directed straight forward, and vertical and horizontal hook position is zero. Important for the development of the interfaces: before the crane performs a transition from arbitrary states to the null-position, it first must put down the objects that the crane is carrying in a safe way.

It is important to know that the hook is realized by a magnet, which can be switched on and off. Whether the robot carries something or not can be derived by measuring the weight of the hook. This weight is zero, if the hook just hangs freely . When standing on the ground or on other objects the measured weight must be negative. While carrying something of *reasonable* weight the weight must be larger than zero.

Finally, we are able to define the maximum speed of each motor. These speed values are positive and between 0 and 100. In general, we can not say which velocity, given in meter per second, will result from the maximum speed of 100. This has to be calibrated on any real crane.

In the following, one can see the type definitions that are used to represent the introduced parameters. One can see, that position and speed data are summarized in a `struct` in order to keep the number of single function calls via CORBA as few as possible.

```
interface ICrane {

   typedef long Position;

   typedef unsigned short Speeds;

   enum MagnetState {ON, OFF};

   struct MaxMotorSpeeds {
       Speeds verticalHook;
       Speeds horizintalHook;
```

```
        Speeds arm;
    };

    struct CranePositions {
        Position verticalHook;
        Position horizontalHook;
        Position arm;
    };
    ...
};
```

4.4.1 Configuration and calibration of the real robot

Using the real KURT3D, the crane configuration will be done by the function call `void resetCrane()`. This reset invokes the calibration method and drives the robot in its null-position. We have already defined above, what null-position for the crane means. The maximum motor speed for each motor is set to 100.

The calibration must be finished before other CAN bus messages can be executed, since the micro-controller ignores all CAN bus messages in the calibration mode. Please notice, that the basic-control of the KURT3D is controlled via CAN bus messages as well.

Calling the calibration-method the hook moves up and backward, until it reaches the zero position and stops. After this stop, the crane arm must be manually moved in its 0° position (see Fig. 3(b)) followed by pressing the switcher in front of the cran arm (position 1000 vertical hook movement Fig. 4(a)). This starts the next step of the automatic calibration process, i.e. the calibration of the weight sensor. Now, the hook moves to the horizontal position 1000 and moves the hook down, until the hook is reaching the ground. The changed weight indicates the ground and the downwards movement turns into an upwards movement. Finally, the crane moves to the defined null-position. The reaching of this position indicates the end of the calibration process and the micro-controller leaves the calibration mode, ready for processing new incoming CAN messages.

4.4.2 Getter

Arm and hook position data and maximum motor speeds are represented by the two structs `MaxMotorSpeeds` and `CranePositions`.

The weight of the hook is given as float value. There is no quantitative mapping defined so far. We are only able to measure the weight qualitatively, as introduced above. Weight zero represents a freely hanging hook, while negative values indicate a standing of the hook on the ground or other objects. Weight values larger zero indicate that the hook is carrying something.

Finally, the magnet state can be represented by a variable of the enumeration type `MagnetState`.

```
    void getMaxMotorSpeeds(out MaxMotorSpeeds ms);

    void getCranePositions(out CranePositions positions);
```

```
void getLoadedWeight(out float weight);
boolean isMagnetSwitchedOn();
```

4.4.3 Setter

The magnet will only be switched off, if the current weight of the hook is not larger zero. This situations correspond to a standing of the hook on a, hopefully safe, surface. This boundary condition should ensure, that the robot does not chuck away certain objects, that might lead to some damages of the KURT3D system. The return value of function call `switchMagnet(in MagnetState s)` is `FALSE`, if the state of the magnet was not changed.

Certainly, one is able to set positions for all motors by one function call or by several calls, which drives only one position. If position and speed values are out of the defined range, they are set to the maximum / minimum possible values.

The position can also be incrementally changed. If such an incremental change drives the hook or the arm out of the defined positions, the hook / arm will be stopped at the maximum / minimum possible position.

```
boolean switchMagnet(in MagnetState s);

void setMaxMotorSpeeds(in MaxMotorSpeeds ms);
void setCranePositions(in CranePositions p);

void setHookVerticalPosition(in Position P);
void setHookHorizontalPosition(in Position P);
void setArmPosition(in Position P);

void changeHookVerticalPosition(in Position deltaP);
void changeHookHorizontalPosition(in Position deltaP);
void changeArmPosition(in Position deltaP);

void moveHookDown();
```

A very useful function is `moveHookDown()`. This function call will drive the hook down until it is standing on the ground, i.e. the measured weight is negative.

Notice, the robust, smooth and safe positioning must be implemented by the underlying function. Higher level processes, which use these function calls do not have to consider the final robust control of the positioning. They just set positions, nothing more!

5 Implementation and usage of the interfaces

5.1 CORBA Server for the KURT3D

The implementation of the CORBA server based on the real KURT3D robot platform is stored in the CVS repository on the server gibson. The directory:

`macs/AIS/RCS`

contains at least the following three sub-directories:

```
CorbaClient
CorbaClientSingleORB
CorbaServerKURT3D
```

The directory `CorbaServerKURT3D` contains the CORBA server implementation, that we introduce in this section. One can find class definitions and implementations there:

```
serverKURT3DbaseImpl.cpp
serverKURT3DbaseImpl.h
serverKURT3DcameraImpl.cpp
serverKURT3DcameraImpl.h
serverKURT3DcraneImpl.cpp
serverKURT3DcraneImpl.h
serverKURT3DLaserScannerImpl.cpp
serverKURT3DLaserScannerImpl.h
```

of the defined interfaces for the four main components of our robot system. The file

```
serverKURT3Dall.cpp
```

contains the implementation of a process, that builds up the CORBA server and initializes the services, which provide the functionality of the KURT3D robot system. Certainly, a `Makefile` and the `kurt3d.idl` can be found here, too. The command

```
make
```

compiles the source files and creates the executable:

```
serverKURT3Dall
```

This executable builds up the CORBA server providing the functionality of the real KURT3D system. The script `startServer.sh` (also in this directory) gives an example, how this can be done, (i.e. by using the right arguments, see chapter above).

Important for compiling the sources successfully is the setting of the two variables:

```
AISLIBDIR = -L../../KURT3D_API/lib
AISINCLUDEDIR = -I../../KURT3D_API/include
```

in the given `Makefile`. They refer to the libraries and includes files, which are needed to work with the KURT3D hardware. The current version of these libraries and the corresponding include-files is also stored in the CVS repository. It can be found in

```
macs/AIS/
```

There one can also find examples (incl. Makefiles) for a direct usage of this functionality, i.e. without using CORBA.

A trouble-shooting with respect of compiling the CORBA server should start with checking the settings of these two variables in the Makefile and the location of the KURT3D libraries and includes. Further on, one should check the version of the libraries locally stored or given by the CVS, if needed one has to make an update and / or should contact the partners from FhG/AIS.

5.2 CORBA Client

Other directories can be found in `macs/AIS/RCS` as well. The `CorbaClient`-directory contains many prototypes for client implementations, which should not be used as base for ongoing implementations.

The directory `CorbaClientSingleORB` contains an implementation, which demonstrates how one process (client) can make usage of more than one CORBA server, which provides the services for our KURT3D platform. We used this implementation in order to setup an interactive software environment, which provides the user with current sensor data (camera images) of the robot as well as makes the user able to control the robot via joystick. Further on, the user can switch between the MACSim and a real KURT3D platform.

This implementation is organized as follows. The directory contains the following source files:

```
startClient.sh
kurt3d.idl
Makefile
client.cpp
RcsClient.cpp
RcsClient.h
RunRcsClient.cpp
RunRcsClient.h
```

The first three files `Makefile`, `kurt3d.idl` and `startClient.sh` are well known and will not be explained here again.

Here, we have two class-implementations `RcsClient` and `RunRcsClient`. The first provides the binding of this class with the services provided by the four interfaces of our KURT3D CORBA server.

```
class RcsClient
{
 private:
    CORBA::ORB_ptr orb_ptr_;
    ...

 public:
    RcsClient(CORBA::ORB_ptr orb_ptr);
    RcsClient(int argNmb, char * argContent[],char *strRootPOAName);
    ...

    void openConnectionToServer(char *strCamServer,
                                char *strBaseServer,
                                char *strLaserScanServer,
                                char *strCraneServer);
    void closeConnectionToServer();

    CORBA::ORB_ptr          getORB();
```

```
    Macs::IBasicControl_var getBasisControl();
    Macs::ICamera_var       getCameras();
    Macs::ILaserScanner_var getLaserScanner();
    Macs::ICrane_var        getCrane();
};
#endif
```

This class has two overloaded constructors. The first `RcsClient(CORBA::ORB_ptr orb_ptr)` has a pointer of the CORBA ORB as argument. Without going into the details, this is needed in order to use the same CORBA ORB for the setup of several independent client-processes. The second constructor is used to initialize the CORBA ORB. The first two arguments are organized as the command line argument used in `C`. So basically they contain the same command line arguments given and explained in this document above.

Further on, this class implements open and closing functions, which means the binding with the services by using the symbolic names of the interfaces as they are defined in the CORBA server implementations. Each interface has a symbolic name, which is used as an argument for the connection-method.

One can get an access to the services, the interfaces, by using the pointer given by the last four methods. These pointers can directly be used to invoke the defined methods, specified by the `kurt3d.idl` specification and implemented by the server implementation.

The second class `RunRcsClient` is using such an instance of the `RcsClient` in order to invoke the desired methods by using the corresponding pointers. Further on, the `RunRcsClient` is able to switch between different `RcsClient`s by a simple change of these pointers.

```
class RunRcsClient
{
 private:
    RcsClient* currentRcsClient_;
    ...

    bool singleStep();

 public:

    RunRcsClient();
    ...
    bool switchToClient(RcsClient* newClient);
    ...
    void run(int timeSteps, int mSec);
    void run(int mSec);
};
```

The method `switchToClient(RcsClient* newClient)` is changing to the new client by assigning the corresponding pointer. Certainly, before this is done, the components of the current `RcsClient` must be stopped in a safe way, because all control is gone once the pointer is lost.

```
bool RunRcsClient::switchToClient(RcsClient* newClient){
    stopComponents();
    currentRcsClient_ = newClient;
    return true;
}
```

The actual request to the robot platform is defined in the method `bool singleStep()`. This method is permanently called by the method `void run(int timeSteps, int mSec)` or `void run(int mSec)`. The last one calls `singleStep()` within a time interval larger `mSec` milliseconds. The first "run"-method does the same, but only `timeSteps` times. Both methods are terminated, if the return value of single step is `false`.

```
void RunRcsClient::run(int timeSteps, int mSec){
    bool   goOn = true;
    bool   stopCriteriaOccur = false;
    int    n;

    n=0;
    do{
        if(timer.getTime() > mSec){  // delay
            goOn = singleStep();
            timer.reset();
            n++;
        }
    }while((goOn) && (n < timeSteps));
    return;
}
```

The following example of such a remote method invocation via CORBA in order to get camera images should finish this explanation of the `RunRcsClient`.

```
    ...
    Macs::ICamera::RgbImageArray *rgbImage;

    // read image for the asked camera (LEFT or RIGHT side)
    if((currentRcsClient_->getCameras())->getRgbImage(requestedCam, rgbImage))
        {
            camWin_.loadImage(rgbImage->get_buffer(), 640,480, true);
            camWin_.showImage("cam view");
            ...
            delete rgbImage;
            cout << "camera view refreshed" << endl;
        }
    else
    {
        cout << "got no image data " << endl;
    }
```

The last source file which has to be discussed in this context of CORBA client implementations is named `client.cpp`. Here one can find the binding to the CORBA server:

```
// IAIS component names and initialisations
char *strRootIAIS            = "RootPOA";
char *strNameServiceIAIS     = "NameService";
char *strBaseServerIAIS      = "BasicControlServerIAIS";
char *strCamServerIAIS       = "CameraServerIAIS";
char *strLaserScanServerIAIS = "LaserScannerServerIAIS";
char *strCranServerIAIS      = "CraneServerIAIS";
char *argvRealKurt3D[3] = {"main\0",
     "-ORBInitRef\0",
     "NameService=corbaloc:iiop:localhost:23456/NameService\0"};
int argcRealKurt3D = 3;

// init RcsClient via command line arguments
clientIAIS = RcsClient(argcRealKurt3D,argvRealKurt3D,strRootIAIS);

// initiating  naming service
clientIAIS.initNamingContext(strNameServiceIAIS);

// binding symbolic names to HW devices provided by the KURT3D server
clientIAIS.openConnectionToServer(strCamServerIAIS,
                                  strBaseServerIAIS,
                                  strLaserScanServerIAIS,
                                  strCranServerIAIS);
```

Now, a second RcsClient can be bound to another server providing the same interface by using the same naming context as follows:

```
// init macsim using the  ORB pointers from MACSim
clientMACSim = RcsClient(clientIAIS.getORB());

// init naming context for the MACSim, which is the same
// as for the real KURT3D
char *strNameServiceMACSim     = "NameService";
clientMACSim.initNamingContext(strNameServiceMACSim);
```

Notice, the binding to the second server works only, if these interfaces have different symbolic names. Therefore, the two server implementations must use different symbolic names, although or even because they provide the same interfaces. The `RcsClient` has to know these names in order to create the bindings.

```
// METU-KOVAN MACSim component names and initialisations
char *strBaseServerMACSim      = "BaseServer";
char *strCamServerMACSim       = "CameraServer";
char *strLaserScanServerMACSim = "LaserScannerServer";
char *strCranServerMACSim      = "CraneArmServer";
```

```
// binding symbolic names to the interfaces provided by the MACSim
clientMACSim.openConnectionToServer(strCamServerMACSim,
                                    strBaseServerMACSim,
                                    strLaserScanServerMACSim,
                                    strCranServerMACSim);
```

At this point, we have two instances of the `RcsClient` namely `clientMACSim` and `clientIAIS`. What is needed now, is an instance of the `RunRcsClient` in order to invoke method calls and switch between different servers. The following `while`-loop gives an example how a process can permanently run a given request on different servers, here MACsim and real KURT3D robot.

```
// init RunRcsClient
remoteControl = RunRcsClient();
...

// switch to macsim
remoteControl.switchToClient(&clientMACSim);
do{
    // run macsim until it is terminated
    remoteControl.run(0);
    // switch to real kurt3d
    remoteControl.switchToClient(&clientIAIS);
    // run the same request on the real kurt3d
    // until termination
    remoteControl.run(0);
    // switch to  macsim
    remoteControl.switchToClient(&clientMACSim);
}while(true);
```

At the end of this section, the reader should notice that this example is only a prototype to test a CORBA setup, which enables a user to make usage of the real and the simulated KURT3D robot. All the partners are free to define the client with respect to their special needs and drives.

6 General remarks for the real KURT3D

The usage of CAN bus for the communication with the crane and the base control leads to some constraints, that are very important for all users of the real KURT3D platform.

Please, take care that the calibration of the crane is finished in a defined way (see section of the crane description). Otherwise the crane's micro-controller ignores all incoming commands.

Further on, the CAN bus technique makes it advisable that the micro-controller ignores a sequence of equal commands, which is the case for the current micro-controller firmware of the KURT3D for all set-commands. For instance, a remote method invocation, which incrementally changes the position of the hook by a given and fixed value will not end in a sequence of position changes. The following example:

```
while(true){
   crane->changeHookHorizontalPosition(21);
}
```

drives the hook only by 21 steps. No further position change will happen because, only the first method call will be executed all the other commands are ignored, since they are the same. To overcome this, one might come up with the following alternative:

```
while(true){
   crane->changeHookHorizontalPosition(10);
   crane->changeHookHorizontalPosition(11);
}
```

Now, the sequence of commands is different and the micro-controller will execute each of them.

7 Summary

It has been shown how to set up a generic environment that allows the specification of module interfaces and the test of module communication. This is the simplest possible level of the architectural framework, that has to be further worked out in the deliverables of WP2 – WP5.

Further on, we introduced the IDL interface specification of the KURT3D robot system. This specification describes the basic functionality of the robot. All implementations of "higher" lever functionality must be based on those interfaces. This specification is also used by METU-KOVAN in order to define the API for the MACSim simulation of KURT3D.

We also gave an introduction to the actual implementation of the CORBA servers for the real KURT3D robot systems. An example of an client is introduced as well in order to provide all the partners with a setup to work with the real KURT3D system via CORBA. However, this setup explains also how one client can use the simulation as well as the real hardware, which might lead to a speed-up of systematical tests and experiments in this project.

A IDL specification of the Reference Control System

The following idl file contains the interface specification as it is introduced and discussed above. This corresponding file can be found in the MACS cvs repository on gibson (macs/doc/IDLs/RCS/KURT3Dinterfaces/).

```
/**
 *
 *  IDL Interface Specification KURT3D
 *
 *
 *  Version 0.7
 *  Date: 6.4.2006
 *
 * Authors: Maya Cakmak (maya.ceng.metu.edu.tr),
 *          Martin Huelse (martin.huelse@ais.fraunhofer.de)
 *
 */


#ifndef KURT3D_IDL
#define KURT3D_IDL


module Macs
{
    interface IBasicControl
    {
        ////SPECIAL TYPES FOR BASIC CONTROL INTERFACE////////////////////

        const float    MAX_ROBOT_SPEED = 1.0;          ///< Speed of the robot is limited to this value given in m/s.
        const unsigned short NMB_DISTANCE_SENSORS = 8;  ///< Number of distance (infrered) sensors.
        const unsigned short NMB_BUMPERS          = 6;  ///< Number of bumpers.


        /**
         *  Array that holds the distance sensor readings.
         *  For more details see current version of Deliverable 1.2.2 Specification of Modules
         */
        typedef float DistanceArray[NMB_DISTANCE_SENSORS];

        typedef boolean BumperArray[NMB_BUMPERS]; ///< Array that holds bumper states.


        /**
         *  tiltX can be considered as a rotation about X axis while tiltZ corresponds to a rotation about Z axis
         *  where the axes are taken according to the simulator standard. This means that:
         *      - the tiltZ gives the angle of forward / backward tilt and
         *      - the tiltX represents tilt to  left and right side.
         *  Both values are zero when teh robot is standing on a flat ground.
         *  tiltZ neg. corresponds to an downward tilt with respect to the front of the robot,
         *  pos. values upward respectively.
         *  tiltX represents the tilt to the left or the right. The downward tilt to the left
         *  delivers neg. values, pos. values represent upward tilt of the left side.
         *  (Left and right with respect to the ‘‘robot’s view’’
         *  The values of tilt sensor readings are in terms of degrees and are limited between +90 and -90.
         */
        struct TiltSensors {
            float tiltX;
            float tiltZ;
        };


        /**
         * The whole sensor data is held in one structure.
         */
        struct Sensordata {
            DistanceArray  distance;
            BumperArray    bumper;
            TiltSensors    tilt;
        };


        typedef long Tics;
        typedef double TimeDiff;


        ////FUNCTIONS FOR BASIC CONTROL INTERFACE////////////////////////////////////

        void robotMcReset();    ///< Microcontroller reset function

        void configBasicControl();


        /**
         * Method to set motor control parameters. The parameters are the pulse width modulation
```

```
     * signal, a direction bit and an idle bit:
     *  - PWM signal: Value between 0 and 1024 where 0 is maximum speed and 1024 is full stop.
     *  - Direction bit: 0 for driving forward, 1 for driving backwards.
     *  - Using this function the robot steers the wheels.  I.e. if the robot is driving with
     *  a speed of let's say 50cm/s and it is stopped by setting a PWM signal of 1024 to both
     *  wheels the robot will use its motors to stop immediately.
     *
     *  Important, after xxx ms without sending a pwm signals to the motor via this function
     *  call both motors are stopped. Hence, for our examples we call every 10 ms this function
     *  in order to keep the robot moving.
     *
     *  Further on, it is needed to call this function wait at least 10 ms before calling this
     *  function again, the can bus  probably will stuck!
     *
     * \param nLeftPwm pwm value of the left wheels [0-1024]
     * \param nLeftDirection direction of the left wheels [0,1]
     * \param nRightPwm pwm value of the right wheels [0,1024]
     * \param nRightDirection direction of the right wheels [0,1]
     * \return EnumCanError denoting if the CAN communication was successful.
     *
     */
    boolean setPwmSignals(in long nLeftPwm, in long nLeftDirection, in long nRightPwm, in long nRightDirection);


    /**
     * Method reads the current (relativ) encoder values and determines the time needed for theses
     * values to accumulate.
     * Note: - The encoder values are relative and are send every 10ms by the Kurt robot.
     *               - If (for timing reasons) there arrived more encoder messages, all of them will be read
     *                 and the accumulated value will be returned.
     *               - If there are no new encoder values, tic and time values of 0 will  be returned.
     *               - On the first call of this method the message buffer will be flushed and the result will
     *                 always be 0.
     *
     *  It is needed to call this function wait at least 10 ms before calling this
     *  function again, the can bus  probably will stuck!
     *
     * \param nTicsLeft  left encoder value.
     * \param nTicsRight  right encoder value.
     * \param pldTimeDiff time since the last call (i.e. the time in which these encoder values accumulated).
     * \return - TRUE if encoder are successfully read values, out values are defined, FLASE out values are
     *                   not defined.
     */
    boolean readTics(out Tics nTicsLeft,out Tics nTicsRight, out TimeDiff dTimeDiff);

    /**
     * Gets the sensor data from the KURT base.
     *\param data holds the sensor readings for all the sensors.
     *\param msg is a message used in case of problems. In the simulator these messages are not
     * supposed to be used.
     * \sa getDistanceData
     * \sa getBumperData
     * \sa getTiltData
     * \sa  getMotorSpeeds
     */
    short getSensordata(out Sensordata data, out string msg);


    /**
     * Gets the distance (infra-red) sensor readings.
     *\param distance is a DistanceArray which holds distance readings. Infra-red sensors  are
     * enumarated such that the sensor that faces the back of the robot is the zeroth, and the
     * indexes increase in clockwise direction.
     * \sa DistanceArray
     */
    short getDistanceData(out DistanceArray distance);

    /**
     * Gets the bumper states.
     *\param bumper is a Bumper array that holds the bumper states. These sensors are not
     * currently implemented in the simulator as they have not yet been physically realized.
     */
    short getBumperData(out BumperArray bumper);

    /**
     * Gets the tilt sensor readings.
     * \param tilt holds the reading from the two tilt sensors.
     * \sa TiltSensors
     */
    short getTiltData(out TiltSensors tilt);


};


interface ICamera
{
    //// SPECIAL TYPES FOR CAMERA INTERFACE/////////////////////////////////////////

    /**
```

```
 *  Cameras can have one of the enumarated resolutions which are:
 *   type   horizontal x vertical
 *
 *   VGA   640 x 480
 *   CIF   352 x 288
 *   SIF   320 x 240
 *   QCIF  176 x 144
 *   QSIF  160 x 120
 *   SQCIF 128 x 96
 */
enum CamResolution{ VGA, CIF, SIF, QCIF, QSIF, SQCIF, UNKNOWN};


/**
 * The OMG IDL string type (bounded or unbounded) is mapped
 * to 'char*'. String data are NUL-terminated.
 * The raw image data are given by a bounded array of char.
 * The R/G/B values of a pixel are given as an unsigned character which may have a value between 0 and 255.
 * Therefore the color resolution of a pixel is 3x8 = 24 bit.
 * The length of the length will depend on resolution.
 * For example if the camera is running in VGA-mode, the vector will have 3x640x320 elements.
 * The order of the color data per pixels is r,g,b and the pixels of the image are organized line-by-line.
 */
typedef sequence<octet> RgbImageArray;

/**
 * Sides are enumarated in order to distinguished between the cameras on the left and the right.
 * LEFT and RIGHT are relative to the robot.
 */
enum Side {LEFT, RIGHT};

/**
 *  The positions of the pan and tilt servos holding the camera are held in special types.
 *  These positions are in terms of servo steps. One servo step is approximately 0.23 degrees but
 *  this value may differ with different servos.
 *  Null-position means that both servos are in position zero, which makes the camera look
 *  straight forward.
 *  An increase of the pan value means an orientation to the right while
 *  a decrease means an orientation to the left.
 *  An increase of the tilt value is an upward orientation while
 *  a decrease is a downward orientation.
 */
typedef long Pan;
typedef long Tilt;


//// FUNCTIONS FOR CAMERA INTERFACE  ////////////////////////////////////////////////

/**
 * Configures the camera. The camera should be configured before any of its methods
 * is called.
 * \param s determines which camera is to be configured (right or left).
 * \param resolution is one of the CamResolution values.
 * \param panPos is the initial pan position of the camera.
 * \param tiltPos is the initial tilt position of the camera.
 * \param msg can be used to get status information about the camera.
 * \return value is negative, if the camera does not support the given reslution. This can
 * also be used to code different errors that may occur during configuration.
 */
short configCamera(in Side s,in CamResolution resolution,in Pan panPos,in Tilt tiltPos, out string  msg);

/**
 * Method to learn the current configuration of the camera.
 * \param s determines side of the camera whose configuration will be returned.
 * \param panPos is the current pan position of the camera.
 * \param tiltPos is the current tilt position of the camera.
 * \return value gives the current resultion of  the webcam.
 */
CamResolution getConfigCamera(in Side s, out Pan panPos, out Tilt tiltPos);


/**
 * Sets the pan and tilt of the camera.
 * \param s determines side of the camera whose position will be changed.
 * \param panPos is the desired pan position.
 * \param tiltPos is the desired tilt position.
 * \return value is false if a problem occurs, true otherwise.
 */
boolean setCameraPanTilt (in Side s,in Pan panPos, in Tilt tiltPos);

/**
 * Sets the pan position of the camera.
 * \param s determines side of the camera whose position will be changed.
 * \param panPos is the desired pan position.
 * \return value is false if a problem occurs, true otherwise.
 */
boolean setCameraPan (in Side s, in Pan panPos);

/**
 * Sets the tilt position of the camera.
```

```
     * \param s determines side of the camera whose position will be changed.
     * \param tiltPos is the desired tilt position.
     * \return value is false if a problem occurs, true otherwise.
     */
    boolean setCameraTilt (in Side s, in Tilt tiltPos);

    /**
     * Gets the current pan and tilt position of the camera.
     * \param s determines side of the camera whose position will be returned.
     * \param panPos is the pan position.
     * \param tiltPos is the tilt position.
     */
    void  getCameraPosition(in Side s, out Pan panPos, out Tilt tiltPos);

    /**
     * Incremental change of the positions. (Input values can be positive as well as negative.)
     * An increase of pan position represents a orienation to the right.
     * An increase of tilt position is related to an upward motion of the camera.
     * Null position (i.e., both positions are zero) is defined  as the forward orientation
     * of the camera relative to the robot. The optical axis is parallel to the ‘‘base axis’’ of the robot.
     * Rerturn value is FALSE, if the position could not be reached or if the offset exceeds the servo limits.
     */
    boolean changeCameraPanTilt     (in Side s, in Pan panOffset, in Tilt tiltOffset);
    boolean changeCameraPan   (in Side s, in Pan panOffset);
    boolean changeCameraTilt (in Side s, in Tilt tiltOffset);


    /**
     * Takes a snapshot with the configured resolution and returns the image data.
     * \param s determines side of the camera.
     * \param data is an RgbImageArray which holds the image data.
     */
    boolean getRgbImage(in Side s,out RgbImageArray data);

};



interface ILaserScanner
{
    //// SPECIAL TYPES FOR LASER SCANNER INTERFACE ////////////////////////////////

    /**
     * A laser scan can be done with three different resolutions;
     * 0.25, 0.5 and 1.0 degree. Therefore a full scan of 180 degrees
     * will give 720, 360 and 180 data points with corresponding scan resolutions.
     */
    enum Resolution {quarter, half, one};

    /**
     * A 2D scan can have a specific tilted position.
     * The tilt position of the scanner depends on the servo position, therefore it is in
     * terms of servo steps, which corresponds to approximately 0.23 degrees.
     * In a 3Dscan the Position correnspponds the center of the vertically spanned range.
     */
    typedef long Position;


    /**
     *  Both 2D scan and 3D scan may have a horizontal appex angle.
     *  An apex angle of d means that the scan is made from -d to +d degree, while
     *  zero degree represents the beam going straight forward and d may go up to
     *  90 degrees.
     */
    typedef long ApexAngleHorizontal;

    /**
     *  A 3D scan may also have a vertical apex angle similar to the horizontal,
     *  except that vertical apex must be defined in terms of servo steps.
     */
    typedef long ApexAngleVertical;

    /**
     * In a 3D scan the vertically spanned region may be scanned at different resolutions.
     * In other words the servo may be stepped 1 by 1, 2 by 2 or n by n. This type will give
     * the number of steps between two consecutive 2D scans in a 3D scan.
     */
    typedef long VecticalStep;


    /**
     * The dimention of a 2D scan is the number of data points the the scan will produce.
     */
    typedef short Dimension2D;

    /**
     * The vertical dimention of a 3D scan is the total number of 2D scans made during the 3D scan
     * at consecutive scanner positions. The horizontal dimension is the number of data
     * points taken at each 2D scan of the 3D scan.
     *
```

```
 */
struct Dimension3D {
    unsigned long vertical;
    unsigned long horizontal;
};


/**
 * The data structures to hold the scan data should not have fixed dimensions.
 * In a single 2D scan the number of data point that is delivered, is
 * determined by the resultion and the apex angle.
 */
typedef sequence<double> Tscan2dDistanceArray;
typedef sequence<double> Tscan2dEmissionArray;

/**
 * As the 3D scan can be considered as a sequence of 2D scans performed at different
 * servo positions, the data structure that holds the 3D scan data is a sequence
 * of 2D scan vectors.
 */
typedef sequence<Tscan2dDistanceArray> Tscan3dDistanceArray;
typedef sequence<Tscan2dEmissionArray> Tscan3dEmissionArray;

/**
 * A laser scan may or may not include the emission data. As taking the emission data
 * significantly increases the duration of the scan, it is optionally taken according to
 * the configuration of the scanner. This type holds this information.
 */
typedef boolean EmissionData;


//// FUNCTIONS FOR LASER SCANNER INTERFACE ////////////////////////////////////////

/**
 * Configures the scanner for a 2D scan.
 * \param apex is an ApexAngleHorizontal which defines the horizontal angular range that
 * the scan will cover in degrees. An apex value of d, means a scan from -d to d, where d
 * can not exceed 90 degrees.
 * \param res is a Resolution which can be one, half or quarter.
 * \param pos is the servo position in terms of servo steps. A value of zero corresponds to
 * a scan parallel to the ground. For positive values the scanner will look upwards and for
 * negative values it will look downwards.
 * \param emission determines whether emission data will be taken during the scan.
 * \param msg may be used to hold errors and warnings.
 * \return value is the dimension, i.e. the number of data points that a single scan
 * generates. For expmple for an apex angle of 60 degrees and a half resolution the dimension
 * will be (2*60/0.5) + 1 = 241, since the zero position has taken into account.
 * A dimension value may not be less than 1.
 */
Dimension2D configLaserScanner2D(in ApexAngleHorizontal apex, in Resolution res, in Position pos,
                                 in EmissionData emission, out string msg);

/**
 * Configures the scanner for a 3D scan.
 * \param apexH is an ApexAngleHorizontal similar to a 2D scan. It defines the horizontal
 * angular range that the scan will cover in degrees. An apex value of d, means a scan from
 * -d to d, where d is limited to 90 degrees.
 * \param resH is a Resolution which can be one, half or quarter.
 * \param apexV is a ApexAngleVertical which defines the number of sequantial 2D scans that will
 * occur during the 3D scan.
 * \param verStep gives the number of servo steps between two consecutive 2D scans in a 3D scan.
 * \param pos is the center servo position in terms of servo steps.
 * When the three parameters apexV, verStep and pos are defined we can state that the scan will be
 * from pos-verStep*apexV to pos+verStep*apexV and the total number of 2D scans made during the 3D
 * scan is (2*apexV)+1 (similar to the 2D scan).
 * \param emission determines whether emission data will be taken during the scan.
 * \param msg may be used to hold errors and warnings.
 * \return value is a Dimension3D, whose horizontal resolution the number of data points that a
 * single 2D scan generates and vertical is the number of 2D scans performed.
 */
Dimension3D configLaserScanner3D(in ApexAngleHorizontal apexH, in Resolution resH,
                                 in ApexAngleVertical apexV, in VecticalStep verStep, in Position pos,
                                 in EmissionData emission, out string msg);

/**
 * Makes a 2D scan with the current configuration.
 * Returns false if the scanner had not been configured or in case of failure.
 * 2D scan is made at the given position which may be different from the position to which it was
 * configured initially.
 */
boolean make2dScan(in Position pos);

/**
 * Makes a 3D scan with the current configuration.
 * Returns false if the scanner had not been configured or in case of failure.
 */
boolean make3dScan();


/**
```

```
     * Gets the distance data of the last 2D scan.
     * Return value is false, if no scan had been done previously or in the case that
     * the scan had failed.
     */
    boolean get2dDistance(out Tscan2dDistanceArray scan2dDistanceArray);

    /**
     * Gets the emission data of the last 2D scan.
     * Return value is false, if no scan had been done previously or in the case that
     * the scan had failed or if the scanner was configured with emission=FALSE.
     */
    boolean get2dEmission(out Tscan2dDistanceArray scan2dDistanceArray);

    /**
     * Gets the distance data of the last 3D scan.
     * Return value is false, if no scan had been done previously or in the case that
     * the scan had failed.
     */
    boolean get3dDistance(out Tscan3dDistanceArray scan3dDistanceArray);

    /**
     * Gets the emission data of the last 3D scan.
     * Return value is false, if no scan had been done previously or in the case that
     * the scan had failed or if the scanner was configured with emission=FALSE.
     */
    boolean get3dEmission(out Tscan3dEmissionArray scan3dEmissionArray);


    /**
     * Sets the position of the scanner servo.
     * \param pos is the position to which the scanner will be moved.
     */
    boolean setScannerPosition(in Position pos);

    /**
     * Gets the position of the scanner servo.
     * \param pos is the current position of the scanner servo.
     */
    boolean getScannerPosition(out Position pos);


    /**
     * Checks if the 2D scan has ended.
     * \return value is false while 2D scan is in progress and returns true when it is completed.
     * The scan data can only be accessed when this function returns true.
     */
    boolean is2dScanComplete();

    /**
     * Checks if the 3D scan has ended.
     * \return value is false while 3D scan is in progress and returns true when it is completed.
     * The scan data can only be accessed when this function returns true.
     */
    boolean is3dScanComplete();

};


interface ICrane {


    /**
     *
     *  Vertical and horizontal position of the hook (carrying the magnet) and the
     *  position of the arm (see Del. 1.1.2 for more details and to understand the
     *  differences between arm and hook positions).
     *  The values of horizontal and vertical hook positions are always between 0 and 1000,
     *  even if the corresponding distances between are different. (Position values and
     *  corresponding mapping to the actual location of the hook are also described in the
     *  Del. 1.1.2)
     *
     *  The arm position covers the range from -180 to 180, repesenting the orientation
     *  of the arm in degrees (again see Del. 1.1.2 for details of this mapping).
     */
    typedef long Position;

    /**
     *  Represents the maximal motor speed.
     *  The corresponding values are in the range from 0 to 100.
     *  The valule zero for the maximal speed will lead to  the situations
     *  that the corresponding device is not moving  at all.
     *
     *  The value 100 refers to the maximal motor speeds, which is an
     *  instrinsic parameter of the physical device and so far we can not
     *  say to which resulting hook speed in meter per seconds this lead,
     *  the same holds for the arm position movement.
     *
     */
    typedef unsigned short Speeds;
```

```
/**
 *
 *
 *
 */
enum MagnetState {ON, OFF};

/**
 * To pack the data into one component.
 */
struct MaxMotorSpeeds {
    Speeds verticalHook;
    Speeds horizontalHook;
    Speeds arm;
};


/**
 * ...
 */
struct CranePositions {
    Position verticalHook;
    Position horizontalHook;
    Position arm;
};



/**
 * Resets the crane, which means, crane drives to the Null-position
 * and the  maximal speed values have the default values (i.e. 100).
 *
 * Null-positions means, the magnet is switched off,
 * both (vertical and horizontal) hook positions are zero and
 * the arm orientation is zero degree.
 *
 * Things, which the robot was carrying before this function
 * call was executed, are putted down in a save way,
 * before crane will drive to the  Null-position.
 *
 */
void resetCrane();



/**
 * Functions to get the maximal posible motor speeds and
 * current arm and hook positions in a compact way.
 *
 */
void getMaxMotorSpeeds(out MaxMotorSpeeds ms);
void getCranePositions(out CranePositions positions);


/**
 * Gets the weight of the hook in order to derive
 * is the crane loaded or not.
 *
 * If the hook stand on the ground the weight value
 * must zero.
 *
 */
void getLoadedWeight(out float weight);


/**
 *  Current state of the magnet.
 */
boolean isMagnetSwitchedOn();


/**
 * Swicthes magnet on or off.
 *
 * The magnet will only be switched off, if
 * the current loaded weight is zero.
 * Return value is FALSE, if the state of  the
 * magnet was not changed.
 */
boolean switchMagnet(in MagnetState s);

/**
 * Sets the maximal motor speeds.
 *
 * Speed values out of the range (0 ... 100) results
 * in maximal or minimal speed values.
 */
void setMaxMotorSpeeds(in MaxMotorSpeeds ms);
```

```
        /**
         * Drives the carne (hook and arm) in the position given by the
         * input parameter.
         *
         * In the case that position values are out of the defined
         * range, the final position are set to the minimal or
         * maximal values.
         */
        void setCranePositions(in CranePositions p);

        /**
         * To change to position of single crane components.
         *
         * In the case that position values are out of the defined
         * range, the final position are set to the minimal or
         * maximal values.
         */
        void setHookVerticalPosition(in Position P);
        void setHookHorizontalPosition(in Position P);
        void setArmPosition(in Position P);


        /**
         * Incremental changes of the components.
         *
         * Values can
         * If the deltaP value lead to leave of the range
         * to movement stops at the maximal or minimal
         * value.
         *
         */
        void changeHookVerticalPosition(in Position deltaP);
        void changeHookHorizontalPosition(in Position deltaP);
        void changeArmPosition(in Position deltaP);


        /**
         * Move hook down as long as it is not standing an
         * object or on the ground.
         * This standing is indicated by a loaded weight of zero.
         */
        void moveHookDown();

    };

};

#endif
```

References

[1] OMG. http://www.omg.org/gettingstarted/corbafaq.htm.

[2] Schmidt, D. C. http://www.cs.wustl.edu/~schmidt/TAO.html.