Aibo programming course

July 2005, Summer Course

by Ricardo Téllez
Aibo programming course

- Introduction (1/2h)
- OPEN-R basics (1.5h)
- Sensors (2h)
- Actuators (2h)
- Neural controller (2h)
- Camera (1h)
- Webots simulator (1h)
Introduction
Introduction

- Aibo is a robot dog created by Sony
- Fully programmable
- Several models
Introduction

For the ERS-7 model
18 DOF
Paw sensors (4)
Distance sensors (3)
Touch sensors (4)
Color camera (1)
Stereo mic (2)
Accelerometers (3)
Introduction

- Aibo programs are stored in memory sticks (MS)
- MS are plugged in into Aibo to run the program
- Any type of controller
Introduction

Programming environment by Sony

- R-CODE (scripting language)
- OPEN-R (C++ environment)
- Remote Framework (MSFC environment)
- MEdit (motion editor)
Introduction

Third part programming environments

- **Tekkotsu** (developed at CMU)
  - C++ programming environment on top of OPEN-R

- **URBI** (developed at ENSTA)
  - Powerful scripting that allows remote control of Aibo. Includes C++ libraries for autonomous mode
Introduction

- Example of an URBI program...
- Winner of the Aibo Does Daftpunk contest
Introduction

Setting the environment

- Installing OPEN-R SDK on PC (done)
- Installing base system on MS (done)
  - different wireless modes: basic, wlan or wconsole
  - different memory modes: memprot, nomemprot
- Setting up wireless network (done)
- Compiling & running sample program
- Setting the FTP server
Setting the FTP server

- Compile the TinyFTPD sample program
  
  > make; make install

- Install generated object on MS
  
  > cp TinyFTPD/MS/OPEN-R/MW/OBJS/TINYFTPD.BIN /mnt/usb/OPEN-R/MW/OBJS

- Install the password file on MS
  
  > cp TinyFTPD/MS/OPEN-R/MW/CONF/PASSWD /mnt/usb/OPEN-R/MW/CONF

- Add line to OBJECT.CFG file
  
  > /OPEN-R/MW/OBJS/TINYFTPD.BIN
OPEN-R Basics
OPEN-R Basics

- OPEN-R program composed of **objects** running **concurrently** that **communicate** to each other
- Objects are like processes.
- Objects inherit from OObject.
- Objects are composed of internal states
- Must define virtual functions:
  - DoInit, DoStart, DoStop, DoDestroy
- Example: HelloWord
Compilation Basics

- Example: compilation and execution of the HelloWorld sample program
- Configuration of the Makefile
- Configuration of the HelloWorld.ocf file
- Configuration of the object.cfg file
Compilation Basics

Compiling a sample program

Go to HelloWord program directory:
> cd sample_programs/common/HelloWord

Compile the program:
> make; make install

Transfer the program to the MS:
> cp -r HelloWord/MS/OPEN-R MS/

Insert MS on Aibo and switch it on

Telnet to Aibo to see the result:
> telnet 147.83.60.22x 59000
Objects communicate through gates by using message passing.

- Gates are unidirectional and are identified by a name.

ObjectComm sample program.
Objects communications

- Objects are composed of **internal states**.
- Transitions between states are started by reception of messages from other objects (event oriented programming)

The sender of the message is called the **subject**. The receiver is called the **observer**.

Messages can be of any type of data.

The **Assert READY** (AR) message indicates readiness.
Objects Communication

- Subjects are referred by the `subject[]` array and observers by `observer[]`

To send a message
1. Initialize message’s contents
   \[\text{strcpy}(\text{str}, "!!! Hello world !!!")\];
2. Assign message
   \[\text{subject}[\text{sbjSendString}]->\text{SetData}(\text{str}, \text{sizeof(str)});\]
3. Notify observer
   \[\text{subject}[\text{sbjSendString}]->\text{NotifyObservers}();\]

To receive a message
1. Retrieve message by casting
   \[\text{const char* text} = (\text{const char *})\text{event}.\text{Data}(0);\]
2. Process message
   \[\text{OSYSPRINT}(("\text{SampleObserver::Notify()} %s\n", \text{text});\]
3. Send AR to subject
   \[\text{observer}[\text{event}.\text{ObsIndex}()]->\text{AssertReady}();\]
Objects Communication

The **stub.cfg** file defines the gates of the object. May require Dummy gates

- ObjectName : SampleObserver
- NumOfOSubject : 1
- NumOfOObserver : 1
- Service : "SampleObserver.DummySubject.DoNotConnect.S", null, null

The **connect.cfg** file defines how objects connect to each other

- SampleSubject.SendString.char.S SampleObserver.ReceiveString.char.O

The **object.cfg** contains list of objects to execute

- /MS/OPEN-R/MW/OBJS/POWERMON.BIN
- /MS/OPEN-R/MW/OBJS/SUBJECT.BIN
- /MS/OPEN-R/MW/OBJS/OBSERVER.BIN
Objects Communication

An object’s life

Object initialised: send AR to subjects

Object waits on a state for a message from one of its subjects

When received a message, the object activates a method to process it

When message processed, it sends an AR message to the subject
Virtual Functions

DoInit procedure

- Called when object loaded in memory
- Sets up gates and registers observers and subjects of the object
- Use OPEN-R macros to do the job

```cpp
OSstatus
SampleObserver::DoInit(const OSystemEvent& event)
{
    NEW_ALL_SUBJECT_AND_OBSERVER;
    REGISTER_ALL_ENTRY;
    SET_ALL_READY_AND_NOTIFY_ENTRY;
    return oSUCCESS;
}
```
Virtual Functions

DoStart procedure

Called when DoInit finished in all objects

Sends AR message to all observers

May change from IDLE to another state

Use OPEN-R macros to do the job

```c
OSStatus SampleObserver::DoStart(const OSystemEvent& event)
{
    ENABLE_ALL_SUBJECT;
    ASSERT_READY_TO_ALL_OBSERVER;
    return oSUCCESS;
}
```
Virtual Functions

DoStop procedure

- Called at shutdown of the system
- Sends NAR message to all observers
- May change to IDLE state
- Use OPEN-R macros to do the job

```cpp
OStatus SampleObserver::DoStop(const OSystemEvent& event) {
    DISABLE_ALL_SUBJECT;
    DEASSERT_READY_TO_ALL_OBSERVER;
    return oSUCCESS;
}
```
Virtual Functions

DoDestroy procedure

- Called after DoStop finished in all objects
- Deletes all objects
- Use OPEN-R macros to do the job

SampleObserver::DoDestroy(const OSystemEvent& event)
{
  DELETE_ALL_SUBJECT_AND_OBSERVER;
  return oSUCCESS;
}
Sensors
System Objects

Two system objects

OVirtualRobotComm is in charge of the access to sensors, actuators and camera

OVirtualAudioComm is in charge of all the things related to sound (play and record)

Programmer’s objects must communicate with them in order to obtain sensors and audio values, and to send commands to actuators

They act like any other OPEN-R object
Primitives

Any sensor and actuator is defined by its own **primitive locator**

Ex:

"PRM:/a1-Sensor:a1", \(\text{\# ACCELEROMETER Y}\)

To access a sensor or actuator, a **primitive ID** is required.

Translation from primitive locator to primitive ID is done by the **OPENR::OpenPrimitive** call:

```cpp
for (int i = 0; i < NUM_ERS7_SENSORS; i++)
    result = OPENR::OpenPrimitive(ERS7_SENSOR_LOCATOR[i], &sensorID);
```
Specifying connection

Sensor values are delivered in **frames** of 8ms

Connection to OVirtualRobotComm must be specified.

Create a gate in stub.cfg and define handling proc.

ObjectName : SensorObserver7
NumOfOSubject : 1
NumOfOObserver : 1

Define the connection in connect.cfg.

OVirtualRobotComm.Sensor.OSensorFrameVectorData.S
Sensor information is obtained in a message coming from OVirtualRobotComm through the Sensor gate.

Data obtained is a structure of type OSensorFrameVectorData.
Accessing Values

Steps to access a value (1/2)

A message is received from OVirtualRobotComm. The associated routine is activated.

Get primitive ID of sensor you want to access, and identify it inside the OSensorFrameInfo array

```c
for (int j = 0; j < sensorVec->vectorInfo.numData; j++) {
    OSensorFrameInfo* info = sensorVec->GetInfo(j);
    if (info->primitiveID == sensorID) {
        ers7idx[i] = j;
        break;
    }
}
```
Accessing Values

Steps to access a value (2/2)

Use the index obtained to access the actual values at the OSensorFrameData

Process the data and send the AR message when finished

```cpp
void SensorObserver7::NotifyERS7(const ONotifyEvent& event)
{
    OSensorFrameVectorData* sensorVec = (OSensorFrameVectorData*)event.Data(0);
    if (initSensorIndex == false) {
        InitERS7SensorIndex(sensorVec);
        initSensorIndex = true;    }
    OSYSPRINT("ERS-7 numData %d frameNumber %d\n",
        sensorVec->vectorInfo.numData, sensorVec->info[0].frameNumber);
    PrintERS7Sensor(sensorVec);
    observer[event.ObsIndex()]->AssertReady();
}
```
Accessing Values

void SensorObserver7::PrintSensorValue(OSensorFrameVectorData* sensorVec, int index)
{
    [...]
    
    OSensorFrameData* data = sensorVec->GetData(index);
    OSYSPRINT(("[%2d] val %d %d %d %d\n",
               index,
               data->frame[0].value, data->frame[1].value,
               data->frame[2].value, data->frame[3].value));
    [...]
}

void SensorObserver7::PrintJointValue(OSensorFrameVectorData* sensorVec, int index)
{
    [...]
    OSensorFrameData* data = sensorVec->GetData(index);
    OJointValue* jval = (OJointValue*)data->frame;
    OSYSPRINT(("[%2d] val %d %d %d %d\n", index, jval[0].value, jval[1].value,
               jval[2].value, jval[3].value));
    [...]
}
Actuators
Actuators

Actuator values are delivered in **frames** of 8ms.

Connection to OVirtualRobotComm must be specified.

Create a gate in stub.cfg and define handling proc.

ObjectName : MovingHead7
NumOfOSubject : 1
NumOfOObserver : 1

Define the connection in connect.cfg.

MovingHead7.Move.OCommandVectorData.S
OVirtualRobotComm.Effector.OCommandVectorData.O
Actuators

- Commands to actuators are sent by messages through the **Effector** gate of **OVirtualRobotComm**
- Data sent is a structure of type **OCommandVectorData**
Accessing actuators: steps

- Initializing the system
- Getting primitive IDs
- Setting joints gains (only for joints)
- Calibrating the joints (only for joints)
- Selecting a shared memory region
- Setting the effector value
Accessing actuators

Initializing the motor system (only for joints)

OPENR::SetMotorPower(opowerON)

Getting primitive IDs

static const char* const JOINT_LOCATOR[] = {
    "PRM:/r1/c1-Joint2:11",      // TILT1
    "PRM:/r1/c1/c2-Joint2:12",   // PAN
    "PRM:/r1/c1/c2/c3-Joint2:13" // TILT2
};

OPrimitiveID jointID[NUM_JOINTS];

MovingHead7::OpenPrimitives()
{
    for (int i = 0; i < NUM_JOINTS; i++) {
        OStatus result = OPENR::OpenPrimitive(JOINT_LOCATOR[i],
                                          &jointID[i]);
        
        [...]  
    }
}
Setting Joints Gains

Define joint gains values, usually in *.h

```c
static const word   TILT1_PGAIN        = 0x000a;
static const word   TILT1_IGAIN        = 0x0004;
static const word   TILT1_DGAIN        = 0x0002;
```

Enable and set the gains

```c
void MovingHead7::SetJointGain()
{
    OPENR::EnableJointGain(jointID[TILT1_INDEX]);
    OPENR::SetJointGain(jointID[TILT1_INDEX],
                        TILT1_PGAIN,
                        TILT1_IGAIN,
                        TILT1_DGAIN,
                        PSHIFT, ISHIFT, DSHIFT);
    [...]}
```
Calibrating the Joints

Reads the actual value of the joints and applies it to them for calibration

MovingResult
MovingHead7::AdjustDiffJointValue()
{
    OJointValue current[NUM_JOINTS];
    for (int i = 0; i < NUM_JOINTS; i++) {
        OPENR::GetJointValue(jointID[i], &current[i]);
        SetJointValue(region[0], i,
                      degrees(current[i].value/1000000.0),
                      degrees(current[i].value/1000000.0));
    }

    subject[sbjMove]->SetData(region[0]);
    subject[sbjMove]->NotifyObservers();
    return MOVING_FINISH;
}
Selecting a RCRRegion

First step: creating RCRRegions

- Commands are not directly sent to effectors: we use **shared memory regions** of type RCRRegion.
- RCRRegions act like a buffer, bringing smoothness and allowing large commands.

Steps for the creation of a RCRRegion:

1. Create a command structure
2. Create an RCRRegion assigning the command structure
3. Fill in the RCRRegion with the general parameters required for the command that will contain...
Selecting a RCRegion

First step: creating RCRegions

Create the command structure

```cpp
void MovingHead7::NewCommandVectorData()
{
    result = OPENR::NewCommandVectorData(NUM_JOINTS,
                                           &cmdVecDataID, &cmdVecData);
}
```

Create the RCRegion

```cpp
region[i] = new RCRegion(cmdVecData->vectorInfo.memRegionID,
                         cmdVecData->vectorInfo.offset,
                         (void*)cmdVecData,
                         cmdVecData->vectorInfo.totalSize);
```

Fill the RCRegion with command data

```cpp
    cmdVecData->SetNumData(NUM_JOINTS);
    for (int j = 0; j < NUM_JOINTS; j++) {
        info = cmdVecData->GetInfo(j);
        info->Set(odataJOINT_COMMAND2, jointID[j],
                   ocommandMAX_FRAMES);
    }
```
Selecting a RCRegion

Second step: selecting RCRegions for commands

When required a memory region is selected where to put the effector command.

```cpp
RCRegion* MovingHead7::FindFreeRegion()
{
    for (int i = 0; i < NUM_COMMAND_VECTOR; i++) {
        if (region[i]->NumberOfReference() == 1) return region[i];
    }
    return 0;
}
```
Setting the effector value

After selecting a region, the initial and final values of the joint are calculated. Then, a **user function** is called to perform the actual sequence of frame commands.

```c
RCRegion* rgn = FindFreeRegion();
OSYSDEBUG(("FindFreeRegion()%x \n", rgn));
SetJointValue(rgn, TILT1_INDEX, s_tilt1, s_tilt1 + d_tilt1);
SetJointValue(rgn, PAN_INDEX,   s_pan,   s_pan   + d_pan);
SetJointValue(rgn, TILT2_INDEX, s_tilt2, s_tilt2 + d_tilt2);
```
Setting the effector value

The user function implements the filling of the RCRegion with a sequence of data, indicating the value the effector must have at each frame.

```c
void MovingHead7::SetJointValue(RCRegion* rgn, int idx, double start, double end)
{
    OCommandVectorData* cmdVecData = (OCommandVectorData*)rgn->Base();

    OCommandInfo* info = cmdVecData->GetInfo(idx);
    info->Set(odataJOINT_COMMAND2, jointID[idx], ocommandMAX_FRAMES);

    OCommandData* data = cmdVecData->GetData(idx);
    OJointCommandValue2* jval = (OJointCommandValue2*)data->value;

    double delta = end - start;
    for (int i = 0; i < ocommandMAX_FRAMES; i++) {
        double dval = start + (delta * i) / (double) ocommandMAX_FRAMES;
        jval[i].value = oradians(dval);
    }
}
```
Finally, we put the data (that is in the RCRegion) in the gate that communicates with the OVirtualRobotComm, and notify it that a new command is there.

subject[sbjMove]->SetData(rgn);
subject[sbjMove]->NotifyObservers();
Neural Controller
Description of the problem

- Creation of an OPEN-R program that must make oscillate each joint of one of Aibo’s legs.
- One feedforward neural network for each joint
- Network weights have previously obtained by training in simulation
- Feedforward C++ object and weights for the three nets are provided
Description of the Problem

- Neural nets must be composed of:
  - two input units: 1st indicates present state of joint, 2nd indicates previous step state
  - five hidden units, with bias
  - one output unit, indicating the velocity to apply to the joint
- Time step is $\text{maxNUMFRAMES} \times 8 \text{ ms} = 128 \text{ ms}$
- Use OPENR::GetJointValue function to read sensors
Design steps

1. Generate the configuration files
2. Write virtual functions
3. Calculate primitive IDs
4. Create shared memory regions
5. Set joints gains and calibrate them
6. Create the Ready function
7. Implement the neural controller
8. Create the SetJointValue function
Webots-Aibo session
Webots Control

To remotely control Aibo through Webots, installation of the Webots server on a MS is required.
The control panel allows for:

- Read sensor data from the robot
- Act on the robot effectors
- Reproduce MTN files
We will use the ers7_mimic controller

Use three different files for compilation

The *Makefile* file to compile Webots controller

The *Makefile.sources* file to specify which files should be cross-compiled

The *Makefile.openr* to cross-compile

For cross-compilation type

>make -f Makefile.openr

Objects to install in the MS are at the OPEN-R directory generated in the controller directory
Camera
Camera Data

- Images can be received in three different resolutions (layers)
- An additional layer for color detection
- Images are received in YCrCb format
Camera Data

Camera can be configured in gain, white balance and shutter speed.

Images from Aibo’s camera are received in messages coming from OVirtualRobotComm through the FbkImageSensor gate.
Camera Message

Specify the connection to the virtual object at stub.cfg

ObjectName : ImageObserver
NumOfOSubject : 1
NumOfOObserver : 1
Service : "ImageObserver.DummySubject.DoNotConnect.S", null, null

And connect the objects at connect.cfg

ImageObserver.Image.OFbkImageVectorData.O
Obtaining the layers

Obtain the image message

```cpp
OFbkImageVectorData* fbkImageVectorData = (OFbkImageVectorData*)event.Data(0);
```

And obtain the layers

```cpp
void ImageObserver::SaveRawData(char* path,
                                OFbkImageVectorData* imageVec,
                                OFbkImageLayer layer)
{
    OFbkImageInfo* info = imageVec->GetInfo(layer);
    byte* data = imageVec->GetData(layer);
    size_t size;
    if (layer == ofbkimageLAYER_C) {
        size = info->width * info->height;
    } else if (info->type == odataFBK_YCrCb) {
        size = 3 * info->width * info->height;
    } else if (info->type == odataFBK_YCrCb_HPF) {
        size = 6 * info->width * info->height;
    }
    // Other cases...
}
```