2005年1月末日

卒業制作

AIBOを自由に動かすプログラミング

法政大学
国際文化学部　国際文化学科
４－Ｅ　－１Ｇ０４１５番
重定ゼミ　　廣瀬　綾

目　次

1．はじめに

2．基礎知識

2.1　AIBO とは
2.2　OPEN－R について
2.3　オブジェクト
■　オブジェクト間通信

3．OPEN－R SDK の開発環境

3.1　インストールの手順
3.2　AIBO 無線 LAN 設定

4．OPEN－R プログラミング

4.1　開発の流れ
4.2　コアクラス
4.3　ファイルの配置
4.4　DoInit()，DoStart()，DoStop()，DoDestroy()
4.5　オブジェクト間通信プログラム

5．AIBO を動かす

5.1　OVirtualRobotComm との通信
5.2　OPEN-R API
5.3　CPC プリミティブ

6．実際のプログラム例

7．おわりに

１．はじめに

＜動機＞
ゼミで AIBO を購入し、初めてその動きを見て感動した。まず、多様な動きとかわいらしい音や光に引き込まれた。どのようにしてこれらの動きを作り出し、どのようにして本物の犬らしい動きを表しているのか、興味を抱いたことから始まった。少し調べていくうちに、本当の犬のような間接の動きや、さまざまなセンサー機能の役割を知り、ますます自分で動かしてみたいと思った。
また、山本昌弘教授の勧めもあり、AIBO のプログラミングのためにはｃ＋＋言語を習得できるということで、挑戦してみようと思った。

＜目的＞
プログラミング能力の向上と C++言語の習得を第一の目的とする。これらを通し、プログラミングをする上での基本的な姿勢や考え方を学ぶ。
また、限られた時間の中で、自分なりに計画を立てて作品を仕上げる能力を見につける。
AIBO は、ロボットの中では比較的親しみやすく、一般的に知られてはいるものの、自分でプログラムを作り、動かせるようになった事を知らない人は多い。まずは身近なゼミ生からだけにでも知ってもらい、さらに AIBO に親しみを持ってもらえるようにしたい。

＜卒論概要＞
今回、卒業制作として AIBO のプログラミングを選択したが、そのプログラミング方法の特徴を学ぶのに大変な時間を費やしてしまった。そのため、自分の作成したプログラムだけではなく、この 1 年間を通して学んだ AIBO のプログラミング方法の基礎から説明することにした。学んできた事のすべてがここに凝縮されているので、あえてそうしようと思う。
また、いろいろなプログラムを作るのに挑戦したが、中でも私が一番納得して、きれいにプログラムを作ることができた、AIBO の脚を動かすプログラムの中身を代表して載せた。

2．基礎知識

2.1　AIBO とは

AIBO とは、ソニーの開発したエンターテインメントロボットである。

従来のロボットといえば、一般的には産業用ロボットなどの実用的なものが主流だった。過酷な労働を代行したり、単純な作業を代わりに行ってくれたりするような、人間の役に立つ存在のイメージが強かった。そのイメージを覆し、楽しくおもしろく、人間と共存することを目的としたロボットが AIBO である。

AIBO の大きな特徴としては、自ら情報を取得し、自ら行動をする、「自律行動」ができることだ。AIBO は、単に人間に命令されたことだけを実行するのではなく、自らのおかれている状況を判断し、それに対する的確な行動をとることができる。AIBO は、眼・耳・触覚などのセンサーをはじめ、十数個の関節を動かすアクチュエーター（筋肉に相当するモーター）、活動に必要なエネルギーを供給する電源など、自律ロボットに必要なハードウェアをすべて備えている。また、センサーを使って取得した情報を認識・判断するプログラムや、体験したことを学び個性を育むための学習・成長プログラム、自律行動の源となる本能・感情プログラムなどを内包した市販のソフトウェア（AIBO-ware）で動く。また、これらの機能を駆使して目の前の状況を把握し、その状況から的確な動きをするために必要なハードウェアもすべて備えている。

＜AIBO の感覚＞

- 視覚･･･＜カラーカメラ＞色や形状を感知。
  ＜距離センサー＞赤外線の距離センサーで、段差と壁の危険を察知。
- 聴覚･･･＜ステレオマイク＞周囲の音や人間の声などを感知。
- 感覚･･･＜タッチセンサー・肉球センサー＞
  頭、アゴ、背中のタッチセンサーでスキンシップを判断。肉球センサーで、地に足が着いているかどうかも判断。
- 平衡感覚＜加速度センサー＞重心や傾きを検知。

また、AIBOは人間とのコミュニケーション能力を持っている。人間の伝えたいことを理解し、自分が伝えたいことも表現する。前述のとおり、AIBOは自律行動を通して、動き・ランプ・音声を使って、人間とコミュニケーションをはかることができるロボットである。

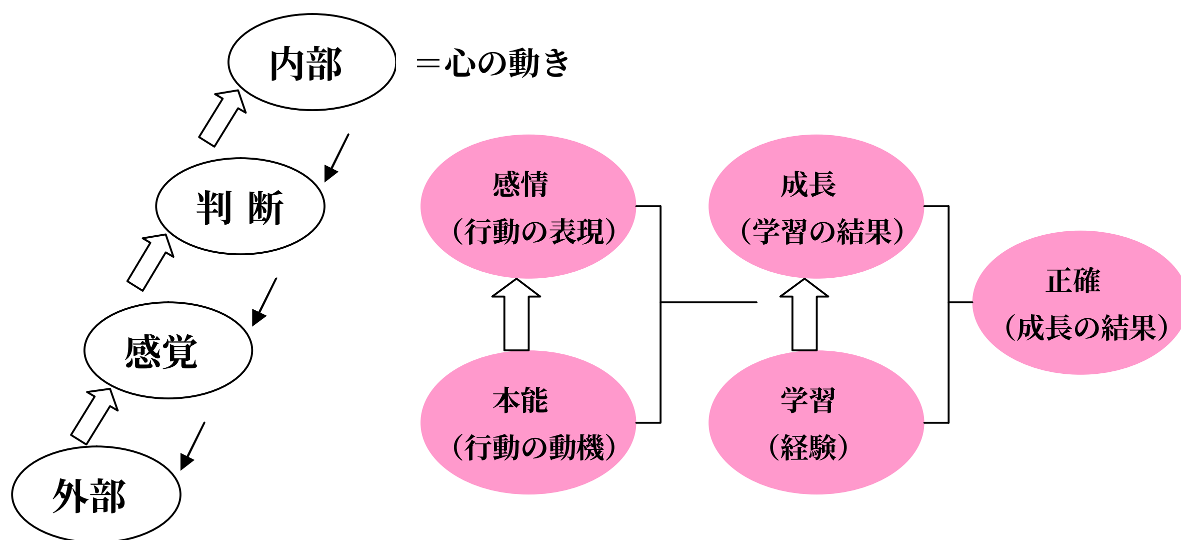
＜AIBOから人へ＞

- 触覚･･･触れられた事を認識。触れ方により意味を理解する。
- 聴覚･･･音声やメロディーを聞き取り、音声認識・音階認識で意味を理解する。
- 視覚･･･物の形や色、距離を読み取り、画像認識でピンクボールや人の顔などを判断する。

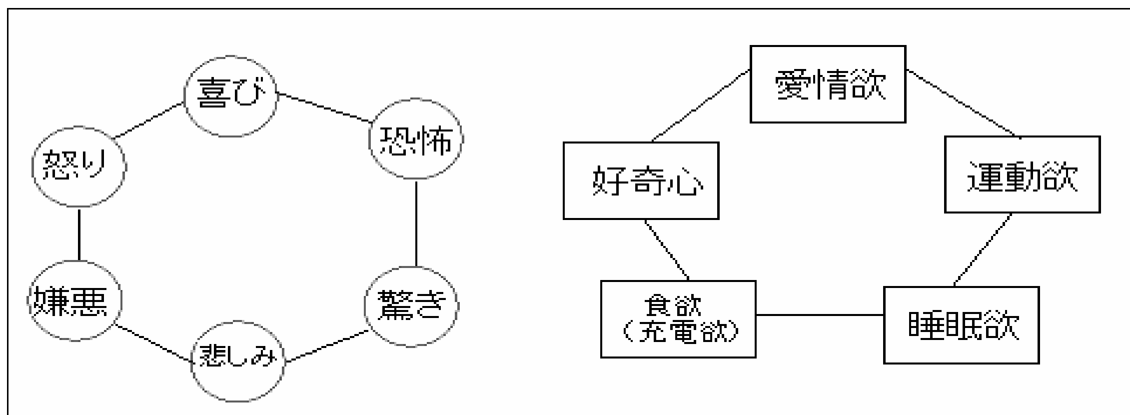
＜人からAIBOへ＞

- 光･･･（目ランプ、尾尻ランプ）光の色やパターンで感情や状態を伝える。
- 音･･･（スピーカー）音やメロディーで感情や状態を伝える。
- ボディーランゲージ･･･（体全体）全身を使った動作で感情や状態を伝える。

AIBOは自律行動型ロボットゆえ、感情を豊かに表現したり、体験から学ぶことでできることが上達したり、時間とともに成長を遂げる。AIBOの自律行動は、センサーなどの「感覚」を通じて取得した情報を認識・判断する“知能”の部分にと、「本能」「感情」「学習」「成長」「性格」などから成る“内部の心の働き”の部分とが、下の図のように連動することによって実現されている。



そして、AIBO の“心”の機能の根本にあるのが「本能」。本能は行動の動機になるもので、「愛情欲」「好奇心」「運動欲」「食欲（充電欲）」「睡眠欲」という 5 つの本能（欲求）を持っている。また、行動を表現するために備えているのが「感情」であり、「喜び」「悲しみ」「怒り」「驚き」「恐怖」「嫌悪」などのさまざまな感情を持つ。



ちなみに AIBO の名前の由来は、

- Artificial Intelligence (AI)　⇒　AI **ロボット**
- Eye**（知覚）をもつロボット**　⇒　Eye + **ロボット**
- **人間と共存＝いつでも行動を共にする相手　⇒　相棒**

からきているらしい。

2.2　OPEN－R について

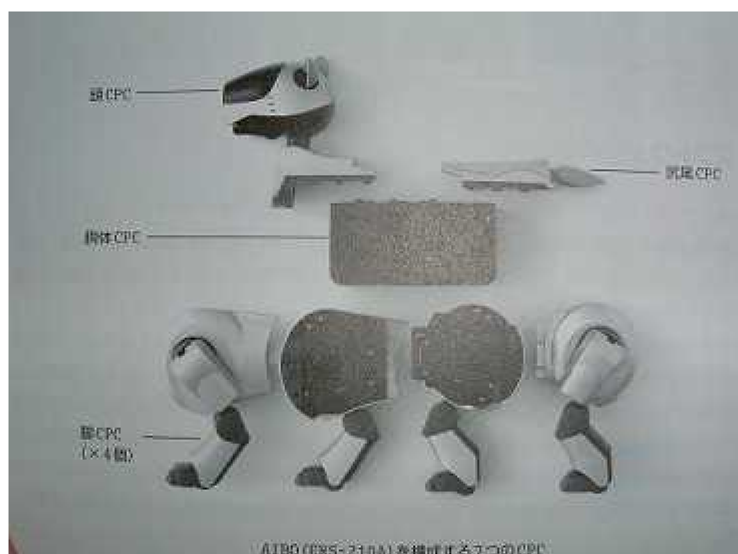
OPEN-R（Open Architecture for Entertainment Robot）とは、1998 年 6 月にソニー株式会社が発表した、エンターテインメントロボットシステムのインターフェイスである。エンターテインメントロボット AIBO にかんするアーキテクチャとして利用されている。これらのインターフェイスは、ロボットのハードウェアやソフトウェアの開発が効率良く行なえるように、階層化・最適化されている。

　また、公開されたソフトウェアの API（Application Program(ming) Interface）を利用した AIBO でのソフトウェア開発環境 OPEN-R SDK も同時に提供されている。OPEN-R SDK では、エンターテインメントロボットのソフトウェア開発に必要な、システム層とアプリケーション層との間のインターフェイス仕様を公開している。

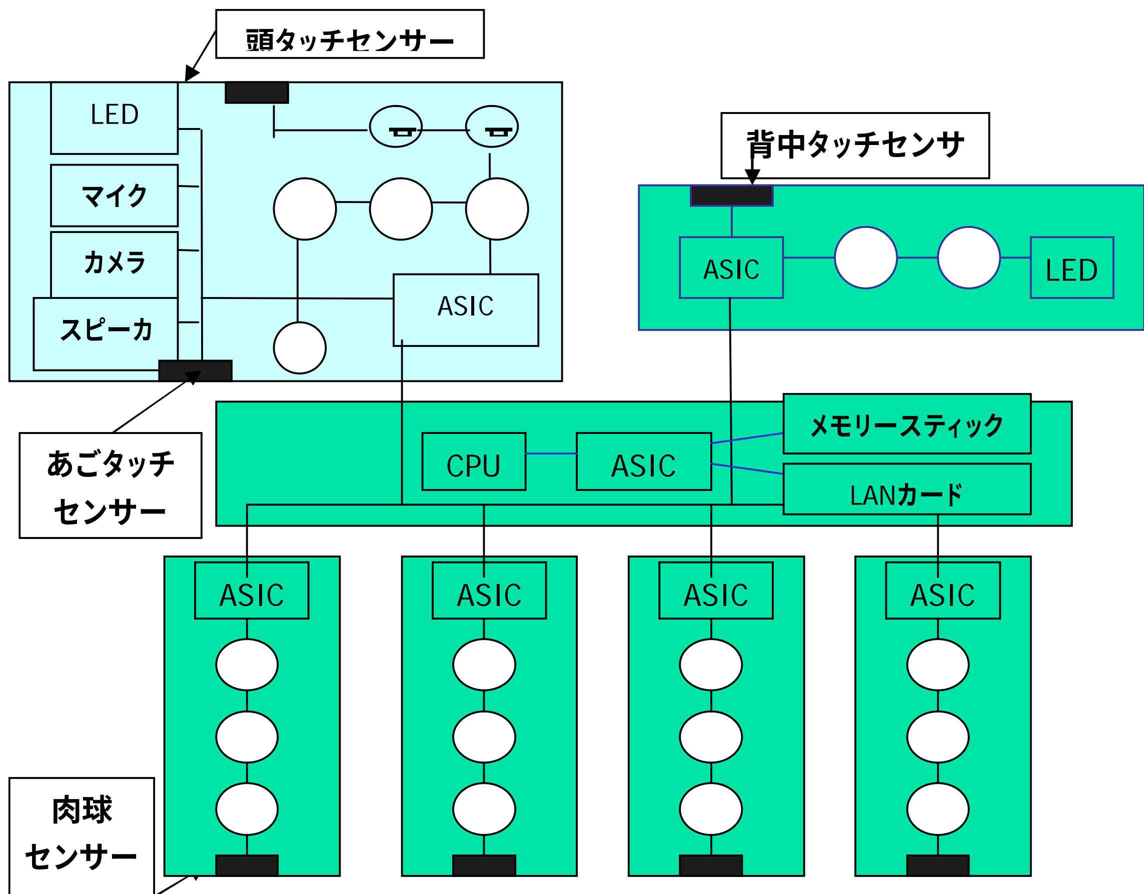
　OPEN-R ソフトウェアには、以下の特徴がある。

＜ソフトウェアの部品化とオブジェクト間通信＞

　OPEN-R では、OPEN-R ハードウェアインターフェイスを持つハードウェア部品のことを、CPC（Configurable Physical Component）と呼ぶ。AIBO（ERS-210A）は、頭・尻尾・脚×４・胴体の７つの CPC で構成されていて、それぞれの CPC は、カメラ・マイク・スピーカ・サーボ機能をもつ関節や各種センサーなど、さまざまなデバイスを備えている。さらに、ソフトウェアが CPC の持つ機能や外形情報などを識別するために必要な情報も含む。



AIBO (ERS-210A) を構成する7つのCPC



OPEN-R ソフトウェアは、オブジェクト指向に基づいて部品化（モジュール化）されている。各モジュールをオブジェクト（OPEN-R オブジェクト）と呼ぶ。OPEN-R では、ロボットを動作させるソフトウェアは、様々な機能を持つ複数のオブジェクトを並行動作させ、オブジェクトが相互に通信（オブジェクト間通信）を行ないながら処理を進める形態で実現する。オブジェクト間の接続関係は、システムが管理するファイルに記述し、起動時にシステムが通信路を確保・設定する。オブジェクト間通信の接続口は、サービス名によって識別するため、部品としての独立性が高く、差し替えが容易になっている。

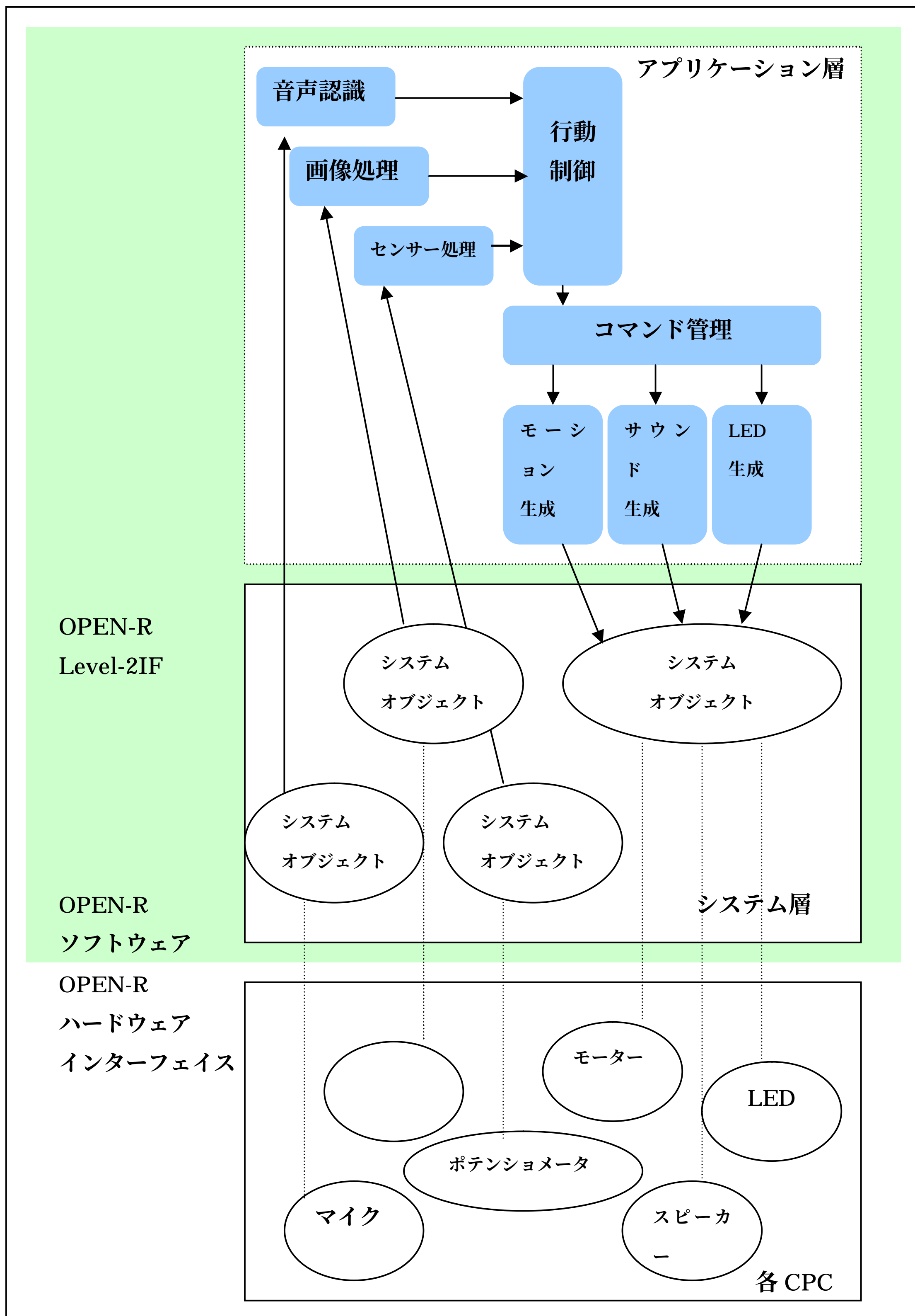
＜ソフトウェアの階層構造とシステム層提供サービス＞

OPEN-R を構成するアプリケーション層とシステム層に関する説明をする。

OPEN-R のシステム層は、大きく分けて２つの機能をもつ。１つは、ロボットの各部位を構成する物理的なデバイスの制御に代表されるロボット自体を扱う機能。もう一つは、ロボットを構成する CPC の管理やアプリケーション層を構成する OPEN-R オブジェクトの生成や消滅の制御といった、管理に関する機能。

これに対し、アプリケーション層は、ロボットにどういった行動をさせるのかを決定する階層で、実現させたい行動に応じて、ロボットの持つ部位の制御・各種センサー情報を利用した認識･行動選択・学習などを OPEN-R オブジェクトとして実装する。

これらシステム層とアプリケーション層間のインターフェイスを OPEN-R Level2 インターフェイスと呼ぶ。このインターフェイスは、サウンドデータ入力、サウンドデータ出力、画像データ入力、関節制御出力、各種センサーデータ入力などのサービスを提供する。このインターフェイスもオブジェクト間通信によって実現されている。これらのサービスを利用すれば、アプリケーション層のオブジェクトは、ロボットを構成するハードウェアデバイスの詳細な知識を持たなくても、ロボットの機能を利用できる。また、システム層は、TCP/IP プロトコルスタックのインターフェイスも提供する。これにより、無線 LAN を利用したネットワーク通信アプリケーションが作成できる。無線 LAN に関しては、第 3 章で詳しく説明する。



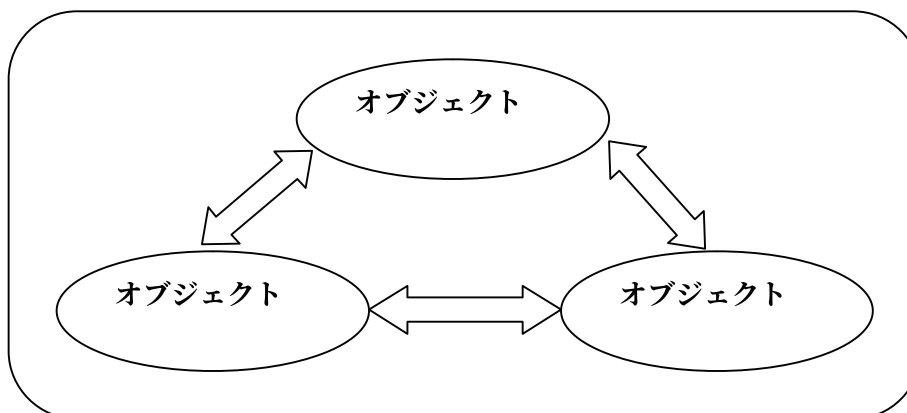
OPEN-R オブジェクト

オブジェクト間通信

ドライバーや ASIC を介した論理的なデータ通信

2.3　オブジェクト

OPEN-R のアプリケーションソフトウェアは複数の OPEN-R オブジェクトから構成される。オブジェクトの概念は、以下の点で Unix や Windows のオペレーティングシステムのプロセスに似ている。



- 一つのオブジェクトには、一つの実行ファイルが対応する。
  オブジェクトは実行時にのみ存在する概念である。各オブジェクトは、コンパイル時に生成される実行ファイルに対応する。実行ファイルは、ソースコードをコンパイル、リンクすることによって作られ、AIBO プログラミングメモリースティック上に置かれる。ロボットの起動時、システムソフトウェアは、実行ファイルをロードし、オブジェクトとして実行を開始する。

- 各オブジェクトは他のオブジェクトと並行して動作する。各オブジェクトは専用の実行スレッドを 1 つ持ち、他のオブジェクトと並行に動作する。

以下はプロセスとは異なるオブジェクト特有の機能。

- オブジェクトはメッセージ通信によって情報を交換する。
  オブジェクトは他のオブジェクトにメッセージを送信することができる。メッセージはデータとセレクターから成る。セレクターは、メッセージの受信側のオブジェクトで実行される処理を特定するための ID である。オブジェクトがメッセージを受信すると、セレクターに対応した関数が呼ばれ、メッセージの中のデータが引数として渡される。セレクターに対応した関数をメソッドと呼ぶ。

オブジェクトの重要な特徴はシングルスレッドであるということだ。これにより、ある瞬間においては、オブジェクトは1個のメッセージのみを処理する。オブジェクトが1個のメッセージを処理中に別のメッセージを受信した場合は、後に受信した方のメッセージはキューに保持され、後で実行される。以下はオブジェクトの処理のサイクルである。

1. システムによってロードされる。
2. メッセージの受信を待つ。
3. メッセージを受信したら、そのメッセージに書かれたセレクターに対応するメソッドを実行する。実行中、場合によっては、他のオブジェクトにメッセージを送信する。
4. メソッドの実行後は2に戻る。

このサイクルは無限ループである。一旦ロードされたオブジェクトは、終了することなく、永続的に存在する。

- オブジェクトは複数のエントリーポイントを持つ。通常のプログラム環境では、プログラムは main()の 1 個のエントリーポイントを保持するが、OPEN-R のオブジェクトは複数のエントリーポイントを保持できる。各エントリーポイントは上述のセレクターに対応付けられる。
  エントリーポイントの一部はシステムによって決められた目的、例えば初期化や終了などの目的に使われる。他のエントリーポイントはオブジェクトに固有の目的に使われる。また、オブジェクトは Prologue()と呼ばれる特別なエントリーポイントを持つ。これはオブジェクトがシステムによってロードされときに1度だけ実行される。Prologue()は初期化と、C++の大域変数のコンストラクタ呼び出しを行う。

■ オブジェクト間通信

エンターテインメントロボットを動作させるソフトウェアは、画像認識・音声認識・行動制御・モーション生成などのさまざまな機能を持つ複数のオブジェクトが相互に通信を行いながら処理を進める。オブジェクト間の通信をOPEN-R ではオブジェクト間通信と呼ぶ。詳しくは4章にて後述。

3．OPEN－R SDK の開発環境

＜AIBO 側で必要なもの＞

- AIBO 本体（ERS－210、ERS－220、ERS－210A、ERS－220A）
  今回は ERS-210 を利用。
- AIBO プログラミングメモリースティック
- AIBO ワイヤレス LAN カード

＜パソコン側で必要なもの＞

- 64MB 以上のメモリと 200MB 以上のハードディスクの空き容量
- パソコンの OS として、Windows XP または Windows2000。MIPS 用開発ツールを自分でビルドすれば、Linux、FreeBSD、Solaris などの UNIX 環境でも利用可能。今回は Windows XP を利用。
- 無線 LAN 環境
- メモリースティックに書き込むための機器
- Cygwin

＜プログラムに必要な知識＞

- C および C++言語の基本
- Make を使ったコマンドライン上での開発
- UNIX の基本的なコマンド

OPEN-R SDK の入手は、OPEN-R オフィシャルウェブサイト（http://www.aibo.com/openr/）からユーザ登録して無料でダウンロードできる。OPEN-R SDK を Windows 上で利用するには、Cygwin が必要になる。Cygwin のオリジナル配布元に関しては（http://cygwin.com）を参照。

3.1　インストールの手順

Windows 環境でのインストールは以下のファイルが必要。

- ✧ cygwin-packages-1.3.17-bin.exe Cygwin 本体
- ✧ mipsel-devtools-3.2-bin-r1.tar.gz MIPS 用開発ツール
- ✧ OPEN_R_SDK-1.1.3-r2.tar.gz　OPEN-R SDK 本体

✧ OPEN_R_SDK-sample-1.1.3-r4.tar.gz サンプルプログラム

インストールの手順は以下の通り。

1. Cygwin のインストール

2. MIPS 用開発ツールのインストール
   Cygwin の/usr/local に mipsel-devtools-3.2-binr1.tar.gz を展開する
   $cd /usr/local
   $tar zxvf mipsel-devtools-3.2-bin-r1.tar.gz

3. OPEN-R SDK のインストール
   2．と同様に/usr/local に展開する
   $cd /usr/local
   $tar zxvf OPEN_R_SDK-1.1.3-r2.tar.gz

4. サンプルプログラムをワークディレクトリにインストール
   $cd ~/prog/openr
   $tar zxvf OPEN_R_SDK-sample-1.1.3-r4.tar.gz

5. システムプログラムのコピー
   /usr/local/OPEN R SDK/OPEN R/MS/WCONSOLE/memprot/OPEN-R をメモリースティックにコピー

3.2 AIBO の無線 LAN 設定

メモリースティック上の OPENR/SYSTEM/CONF/WLANDFLT.TXT をコピーして、WLANCONF.TXT というファイル名に変更し、ファイル内容を編集する。

＜初期設定＞
HOSTNAME=AIBO・・・AIBO のホスト名

ETHER_IP=10.0.1.100・・・AIBO の IP アドレス
ETHER_NETMASK =255.255.255.0・・・LAN のネットマスク
IP_GATEWAY=10.0.1.1･･･LAN のゲートウェイ IP アドレス
ESSID=AIBONET・・・無線 LAN の ESS-ID
WEPENABLE =1・・・無線 LAN で WEP を使う場合 1、使わない場合 0
WEPKEY=AIBO2・・・無線 LAN の WEP キー
APMODE=2・・・無線 LAN の動作モード（0:アドホック、1:アクセスポイント、2:アクセスポイントがなければ，アドホックで接続）
CHANNEL=3・・・無線のチャンネル

[WLANCONF.TXT の例]

```
HOSTNAME=AIBO
ETHER_IP=192.168.0.5
ETHER_NETMASK=255.255.255.0
IP_GATEWAY=192.168.0.1
ESSID=AIBONET
WEPENABLE=1
WEPKEY=AIBO2
APMODE=1
CHANNEL=3
```

４．OPEN－R プログラミング

4.1　開発の流れ

1. オブジェクトの設計
   新規に作成するオブジェクトの機能と、オブジェクト間のデータの流れを設計する。

2. オブジェクト間通信のデータ型の設計
   他のオブジェクトと通信するためのデータ型を設計する。

3. stub.cfg の記述
   オブジェクトが外部からメッセージを受け取るためのエントリーポントと、オブジェクト内に実装されたコアクラスのメンバ関数との接続を、フォーマットに従って stub.cfg(Stub Configuration)ファイルに記述する。stub.cfg には、他のオブジェクトとデータを送受信するためのサービスも記述する。stubgen2 コマンドを実行することで、stub.cfg からコンパイル時に必要な中間ファイルが生成される。

4. コアクラスの実装
   オブジェクトのコアクラスを実装する。stub.cfg で指定したメンバ関数や、コアクラスに必ず実装する 4 個の Doxxx() メンバ関数や、その他のメンバ関数も実装する。

5. ocf のコンフィグレーションの設定
   オブジェクトの実行時のコンフィグレーションを設定する。

6. ビルド
   必要なライブラリをリンクしてビルドする。

7. 設定ファイルの編集
   実行時に必要な設定ファイルを編集する。例えば以下のものがある。
   - OBJECT.CFG・・・実行するオブジェクトを列挙。
   - CONNECT.CFG・・・オブジェクト間の接続を記述。

- DESIGNDB.CFG・・・オブジェクトが実行時にアクセスするファイルをパス付きで記述する。

8. AIBO での実行

AIBO プログラミングメモリースティックの所定の場所に、以下のファイルをコピーする。

- システムファイル等が置かれている OPEN-R ディレクトリ
- 実行ファイル（.BIN）
- 編集した設定ファイル
- モーションやサウンドなどのデータファイル

無線 LAN 環境を構築して AIBO と PC を接続し、AIBO プログラミングメモリースティックを AIBO に挿入して、AIBO を起動する。

9. デバッグ

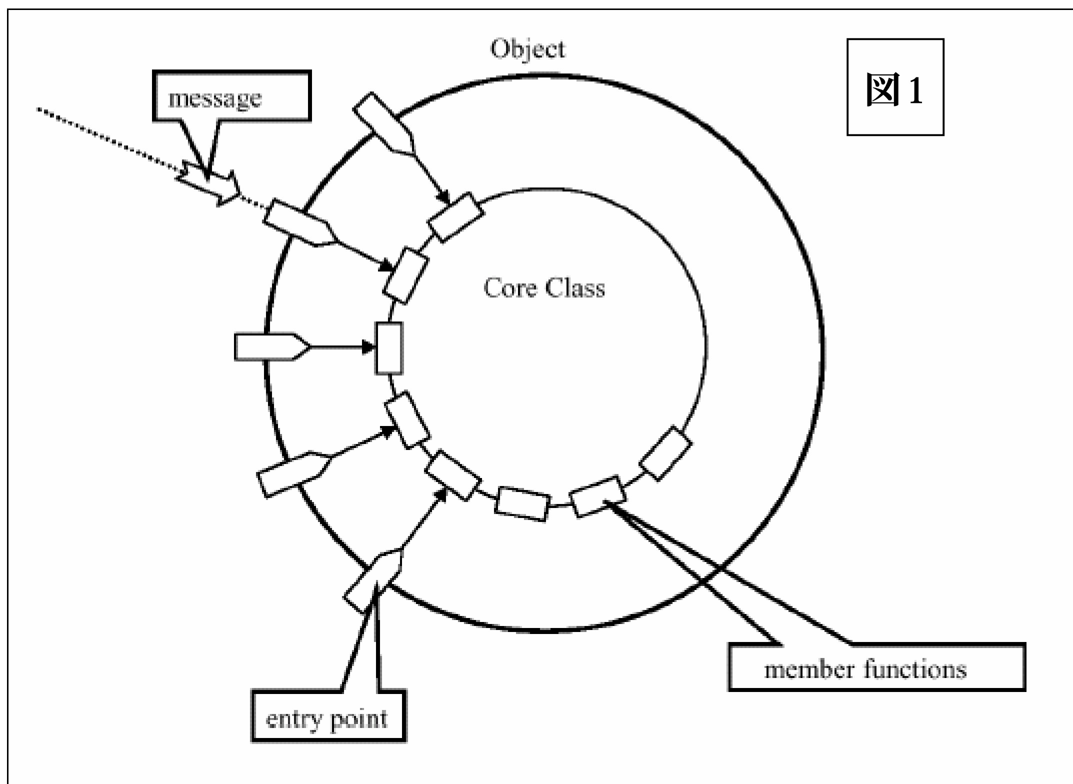
無線 LAN コンソールに表示されるメッセージの中から、ソースコードに記述した OSYSPRINT や OSYSLOG1 などのデバッグ文や、その他のエラー情報を参考にして、バグの原因を調査してデバッグする。

4.2 コアクラス

コアクラスとはオブジェクトを表現する C++のクラスである。１つのオブジェクトにつき、一個のコアクラスだけを定義できる。オブジェクトはメッセージの受信用に複数のエントリーポイントを保持する。下の図１のように、各エントリーポイントはオブジェクトのメソッドに対応し、メソッドはコアクラスのメンバ関数に対応する。

コアクラスには以下の特徴がある。

- ✧ コアクラスは OObject を継承する
- ✧ コアクラスは、DoInit(), DoStart(), DoStop(), DoDestroy()を実装する。
- ✧ コアクラスは、OSubject と OObserver を必要数だけ保持する。
- ✧ コアクラスのメンバ関数は、特定のメソッドと対応するものがある（Ready，Notify など）。



4.3　ファイルの配置

下記図 2 より、Object1、Object2、PowerMonitor にはソースファイル(xxx.cc・xxx.h)、補助的なファイル、メイクファイルが含まれ、MS にはメモリースティックにコピーするための実行ファイルや設定ファイルが含まれる。

メイクファイルは、ソースファイルをコンパイルし、実行ファイル（BIN）を WorkDirectory/MS/OPENR/MW/OBJS の下にコピーするためのメイクファイル。オブジェクト PowerMonitor は、ソニーが提供したプログラムで、AIBO の電源スイッチ（ポーズスイッチ）を監視してシャットダウン時に電源を落とすプログラムである。必ず自分の Work の下に入れる。

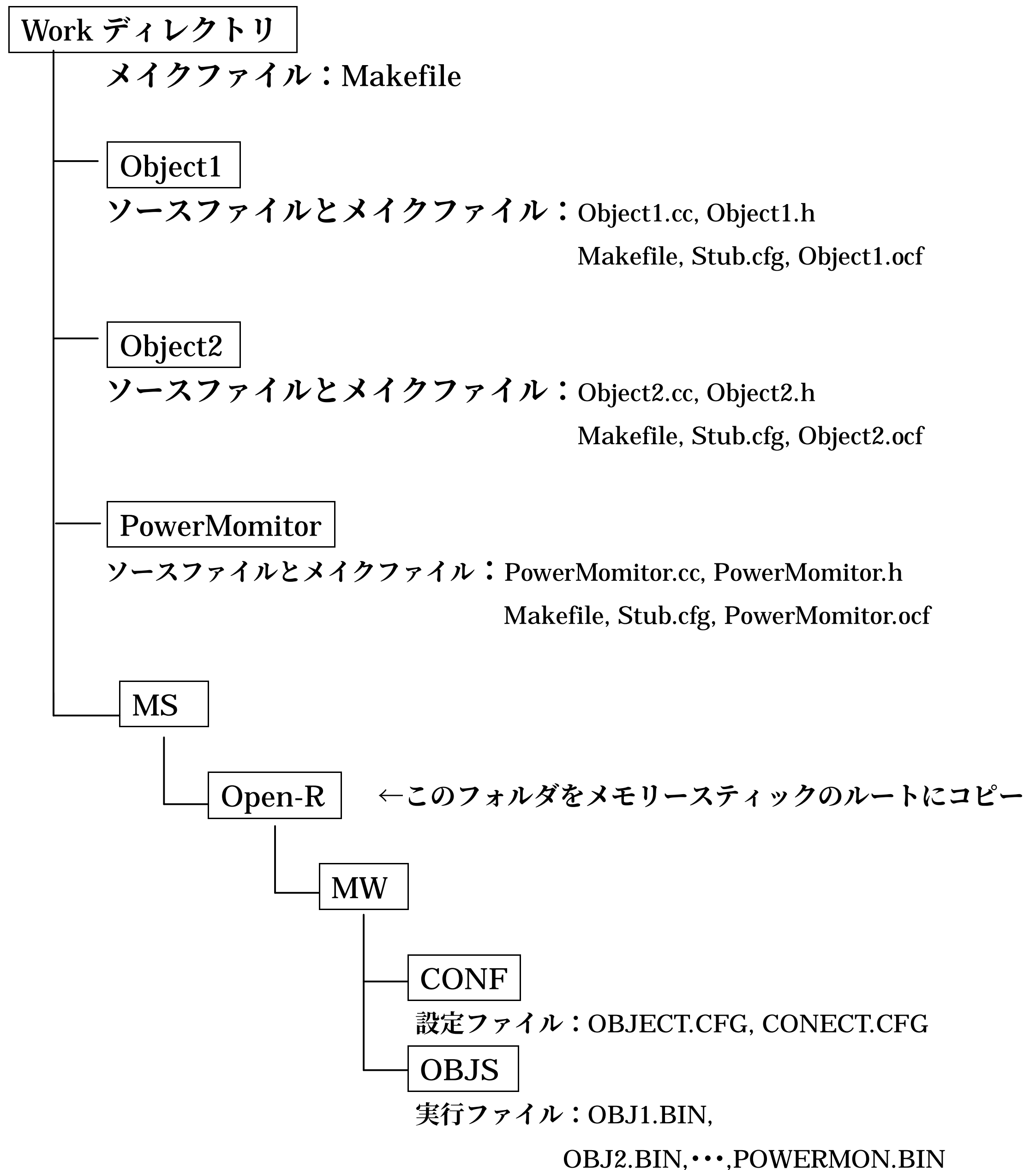
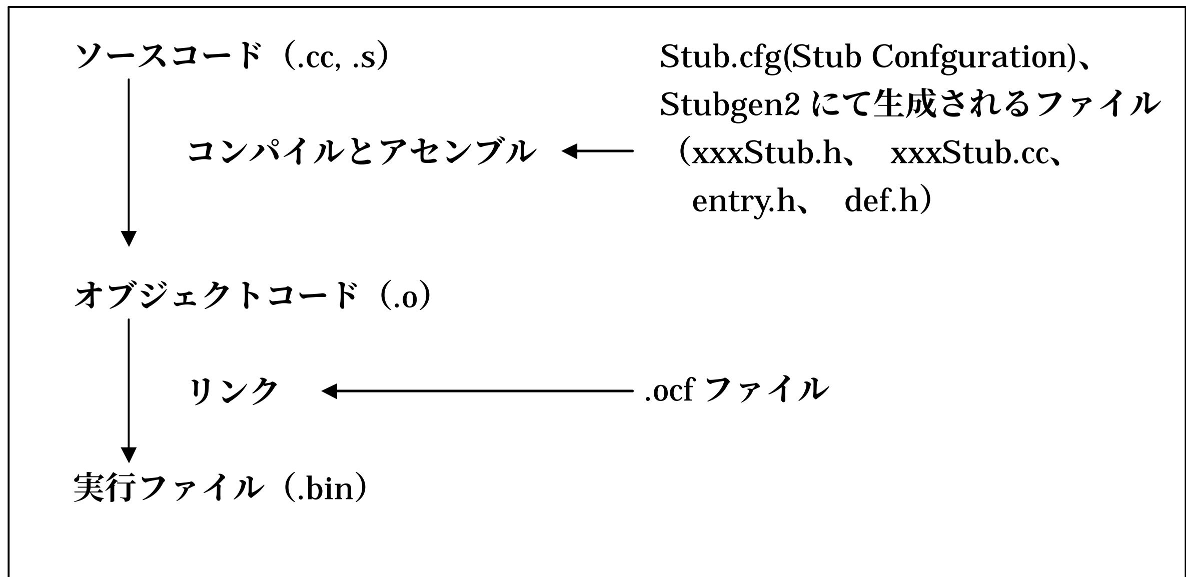


図２　ファイルの配置

＜Makefile＞

以下はオブジェクトをビルドし、実行ファイルを作成する手順。



Makefile の内容は次のようになっている。

[Makefile]

```
PREFIX=/usr/local/OPEN_R_SDK
INSTALLDIR=./MS
CXX=$(PREFIX)/bin/mipsel-linux-g++
STRIP=$(PREFIX)/bin/mipsel-linux-strip
MKBIN=$(PREFIX)/OPEN_R/bin/mkbin
STUBGEN=$(PREFIX)/OPEN_R/bin/stubgen2
MKBINFLAGS=-p $(PREFIX)
LIBS=-lObjectComm -lOPENR
CXXFLAGS=
        -O2
        -g
        -I.
        -I$(PREFIX)/OPEN_R/include/R4000
        -I$(PREFIX)/OPEN_R/include
```

```
.PHONY: all install clean
all: object1.bin
%.o: %.cc
$(CXX) $(CXXFLAGS) -o $@ -c $^
#オブジェクト間通信を行う場合
Object1Stub.cc: stub.cfg
$(STUBGEN) stub.cfg
object1.bin: Object1Stub.o Object1.o object1.ocf
$(MKBIN) $(MKBINFLAGS) -o $@ $^ $(LIBS)
$(STRIP) $@
install: object1.bin
gzip -c object1.bin >
$(INSTALLDIR)/OPEN-R/MW/OBJS/OBJ1.BIN
clean:
rm -f *.o *.bin *.elf *.snap.cc
rm -f Object1Stub.h Object1Stub.cc def.h entry.h
rm -f $(INSTALLDIR)/OPEN-R/MW/OBJS/OBJ1.BIN
```

make コマンドにパラメータを指定しないときは、mipsellinux-g++コマンドで各ソースファイル（.cc ファイル）をコンパイルし，mkbin コマンドで中間ファイルをリンクして実行ファイル（object1.bin）を生成する。

パラメータに「install」を指定すると、object1.bin をメモリースティック用のフォルダ（MS/OPEN-R/MW/OBJS/）に圧縮してコピーする。BIN のファイル名は、ファイル名８文字＋ピリオド＋3 文字までという制限がある。
パラメータ「clean」を指定すると、中間ファイルと実行ファイルを削除する。

✧ 設定ファイルと補助的なファイル

OPEN-R オブジェクトを作成するとき、ソースファイルとメイクファイルの他、作成しなければならない設定ファイルと補助的なファイルは以下のようになっている。

＜OBJECT.CFG＞

　OBJECT.CFG は、AIBO に読み込ませる実行ファイルのリストを記述した設定ファイルである。図 4 では、OPEN-R オブジェクトが Object1 と Object2 と PowerMonitor の 3 つの実行ファイルを読み込ませるため、次のように 3 つのファイルのフルパスを改行で区切って並べる。このパスは AIBO から見たメモリースティック上のパスなので、/MS/で始める。（メモリースティックのルートディレクトリを/MS/としたもの）。OBJECT.CFG の内容は次のようになっている。

```
[OBJECT.CFG]
/MS/OPEN-R/MW/OBJS/POWERMON.BIN
/MS/OPEN-R/MW/OBJS/OBJ1.BIN
/MS/OPEN-R/MW/OBJS/OBJ2.BIN
```

＜OCF ファイル＞

　OCF ファイルは、実行ファイル（オブジェクト）に関する情報を記述する設定ファイル。次のような書式のテキストファイルになる。

object OBJECT_NAME STACK_SIZE HEAP_SIZE SCHED_PRIORITY CACHE TLB MODE

PowerMonitor に含まれる powerMonitor.ocf は、次のようになっている。

[powerMonitor.ocf]
object powerMonitor 3072 16386 128 cache tlb user

＜OBJECT NAME＞

オブジェクト名を指定する。

＜STACK SIZE＞

　スタックのサイズを指定する。スタックは、ローカル変数の確保や関数の呼び出し時に使われる領域。

＜HEAP SIZE＞

ヒープが拡張される際の増分のサイズを指定する。ヒープは、malloc()関数やnew 演算子で確保されるメモリに使われる領域。ヒープは足りなくなると自動的に拡張される。

＜SCHED PRIORITY＞

オブジェクトのスケジューリング、つまり複数のオブジェクトを同時に動かす際に与えられる時間の優先順位。8 ビットの符号なし整数で指定する。上位 4 ビットが大きくなるほど、他のオブジェクトよりも優先順位が高くなる。上位 4 ビットが同じ優先順位の場合は、下位 4 ビットが大きいほどオブジェクトに多くの時間が割り当てられる。推奨値は上位 4 ビットが 8, 下位 4 ビットが 0、つまり 128 である。

＜CACHE＞

「cache」か「nocache」を指定する。「nocache」を指定すると実行時にキャッシュメモリが使えなくなる。通常は「cache」を指定。

＜TLB＞

「tlb」か「notlb」を指定する。「tlb」を指定すると、オブジェクトが使うメモリは仮想アドレスとなり、メモリ保護が働く。モードを「user」にした場合は、「tlb」を指定する。

＜MODE＞

「kernel」か「user」を指定する。「user」を指定するとオブジェクトはユーザモードで、つまり通常のアプリケーションとして実行される。「kernel」を指定するとカーネルモードで、つまり OS と同レベルの権限を持ったプログラムとして実行される。通常は「user」を指定。自分でプログラムを作成する際には, サンプルプログラムの helloworld.ocf をコピーして、オブジェクト名の部分だけを変更すれば良い。また、OCF ファイルのファイル名は適当な名前で良いが、Makefile のなかで記述する OCF ファイル名を作成した OCF ファイルのファイル名に合わせる。

オブジェクト間通信がある場合、CONNECT.CFG と stub.cfg も作成しなければならないが、詳しい内容は、オブジェク間通信のところで説明する。

4.4　DoInit()，DoStart()，DoStop()，DoDestroy()

オブジェクトを表すクラスは、ヘッダファイルの中で次のように OObject クラスを継承し、DoInit()、DoStart()、DoStop()、DoDestroy()の４つのメンバ関数を宣言したものになる。

以下のサンプルプログラム（HelloWorld）は、telnet で AIBO へ接続し、「Hello World」という文字列を表示させるプログラムである。
※ HelloWorld では、オブジェクト間通信を行わないため、stub.cfg と CONNECT.CFG を記述するはありません。

```
[HelloWorld.h]
#include <OPENR/OObject.h>
class HelloWorld : public OObject { //継承
public:
HelloWorld();
virtual ~HelloWorld() {}
virtual OStatus DoInit (const OSystemEvent& event);
virtual OStatus DoStart (const OSystemEvent& event);
virtual OStatus DoStop (const OSystemEvent& event);
virtual OStatus DoDestroy(const OSystemEvent& event);
};
```

OPEN-R プログラミングでは、main()関数に書かれたプログラムが順に実行されるのではなく、OObjec を継承したクラスのメンバ関数が「何々したとき」に呼び出される形になる。この「何々したとき」のことをイベントと呼ぶ。また、基本的な４つのメンバ関数は、次のように呼び出される。

＜DoInit()＞
AIBO が起動したときに呼び出される。通常 DoInit()では、オブジェクト間通信の準備を行う。そのほか、CPC プリミティブを開いたり、AIBO のモーター電源をオンにしたりする。

＜DoStart()＞

DoStart()は、すべてのオブジェクトで DoInit()が呼び出されたあとに呼び出される。通常 DoStart()では、オブジェクト間通信を開始し、プログラムを実際に動作させ始める。

＜DoStop()＞

DoStop()は、AIBO がシャットダウンされるときに呼び出される。通常 DoStop()では、オブジェクト間通信を終了させる作業を行う。

＜DoDestroy()＞

DoDestroy()は、すべてもオブジェクトで DoStop()が呼び出されたあとに呼び出される。通常 DoDestroy()では、オブジェクト間通信に使ったメモリなどの後始末を行う。

ソースプログラムでは、次のように各メンバ関数を定義する。引数として渡される OSystemEvent は、イベントに関する情報を含むクラス。戻り値となる OStatus は整数値で、通常は成功を表す定数 oSUCCESS を返す。

```
[HelloWorld.cc]
#include <OPENR/OSyslog.h>
#include "HelloWorld.h"
HelloWorld::HelloWorld ()
{ }
OStatus
HelloWorld::DoInit(const OSystemEvent& event)
{
//オブジェクト間通信の準備
return oSUCCESS;
}
OStatus
HelloWorld::DoStart(const OSystemEvent& event)
{
//オブジェクト間通信の開始と動作実行
OSYSPRINT(("!!! Hello World !!!  n"));
```

```
return oSUCCESS;
}
OStatus
HelloWorld::DoStop(const OSystemEvent& event)
{
//オブジェクト間通信の終了
return oSUCCESS;
}
OStatus
HelloWorld::DoDestroy(const OSystemEvent& event)
{
//後始末
return oSUCCESS;
}
```

✧　文字列の出力

HelloWorld では、次のマクロを使って無線コンソールに文字を出力する。これらは、OPEN-R SDK のヘッダファイル OPENR/OSyslog.h で定義されている。

OSYSPRINT 文字列を出力
OSYSDEBUG デバック用の文字列を出力
OSYSLOG1 エラーを出力

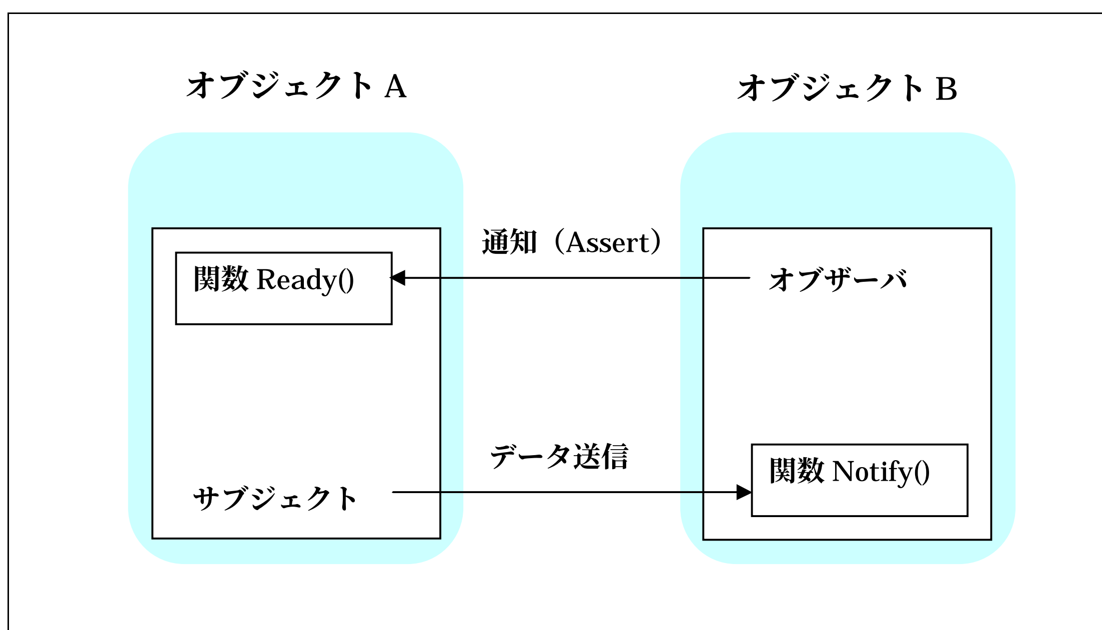
HelloWorld では，マクロ OSYSPRINT を使って文字列「!!!Hello World !!!」を出力する。

4.5　オブジェクト間通信プログラム

オブジェクト間通信を使うと、AIBO の上で動作するアプリケーションを複数のオブジェクトに分割し、互いにメッセージやデータをやり取りすることで、強調して１つのアプリケーションとして動作させる事ができる。

複数のオブジェクトに役割をわけると、それぞれのオブジェクトを独立して開発できるので、開発効率を上げることができる。

OPEN-R でのオブジェクト間通信は、次の図のように各オブジェクトに含まれる「サブジェクト」と「オブザーバ」によって行われる。



サブジェクトは、他のオブジェクトにデータを送信するというサービスを提供する。オブザーバはほかのオブジェクトから送られてきたデータを受け取るというサービスを提供する。オブザーバがサブジェクトからデータを受け取る前には、あらかじめ通知しなければならない。サブジェクトはその通知を受けてオブザーバにデータを送信する。

サブジェクトとオブザーバは、必ずセットで提供されなければならないが、１つのサブジェクトが複数のオブザーバにデータを送ったり、複数のサブジェクトが１つのオブザーバにデータを送ったりすることもできる。

また、オブジェクト間通信は、プログラマが作成したオブジェクトの間で通信をするときに使われるだけではなく、AIBO の手足を動かしたり、AIBO のカ

メラから画像を読み取ったりする時に、システムとやり取りをするにもこのオブジェクト間通信が使われる。そうした場合、プログラマはシステムが持つサブジェクトやオブザーバに対応するオブザーバやサブジェクトを作成することになる。

オブジェクト間通信は、とにかく OPEN-R プログラムの最大の特徴である。SampleSubject と SampleObserver の 2 つのオブジェクトの間で通信を行わせるプログラムを説明するとさらにわかりやすいので例とする。SampleSubject は SampleObserver に「!!!Helloworld!!!」、「!!!Hello world again!!!」という文字を送り、SampleObserver は文字列を受け取って表示するプログラムである。

✧ CONNECT.CFG とサービス名

オブジェクト間通信を行うには、AIBO のシステムに対して、どのサブジェクトがどのオブザーバと通信するのかを指定する必要があり，この対応関係を示すために、CONNECT.CFG という設定ファイルを記述する。

CONNECT.CFG の書式は次の通り。オブジェクト間通信を行うサブジェクトとオブザーバが複数ある場合、改行によって対応関係をいくつでも記述できる。

[CONNECT.CFG の書式]
サブジェクトのサービス名対応するオブザーバのサービス名。

サービス名は、サブジェクトやオブザーバを識別するための名前で、次のようにピリオドで区切った書式で命名する。

[サービス名の書式]
オブジェクト名．サブネーム．データ名．サービスのタイプ（S または O)。

オブジェクト名は通常サブジェクトやオブザーバを含むオブジェクトの名前で、サブネームがサブジェクトやオブザーバに固有の名前になる。データ名は任意だが、どんなタイプのデータを送受信するのかがわかるように付けたほうが良い。サービスのタイプは、サブジェクトの場合は S に，オブザーバの場合は O になる。CONNECT.CFG は、OBJECT.CFG と同じディレクトリ MS/OPEN-R/MW/CONF に置く。

[CONNECT.CFG の例]
SampleSubject.SendString.char.S SampleObserver.ReceiveString.
char.O

上記の例では、オブジェクト SampleSubject のサブジェクト SendString がオブジェクト SampleObserver のオブザーバ ReceiveString と通信することを示している。

✧　オブジェクト間通信のためのメンバ関数

　オブジェクト間通信を行う際には, 4 つの基本メンバ関数（DoInit(),DoStart(), DoStop(),DoDestroy()）の他に、次の Control()、Ready()、Connect()、Notify()の 4 つのメンバ関数をクラスに実装する必要がある。各メンバ関数には任意の関数名を付けらる。

＜Connect()＞

オブザーバを含むオブジェクトのクラスで実装し、サブジェクトとの接続を確立するための関数。

＜Control()＞

サブジェクトを含むオブジェクトのクラスで実装し、オブザーバとの接続を確立するための関数。

＜Ready()＞

サブジェクトを含むオブジェクトのクラスで実装し、対応するオブザーバが通信の準備かできたことをサブジェクトに通知したときに呼び出される。

＜Notify()＞

オブザーバを含むオブジェクトのクラスで実装し、対応するサブジェクトがオブザーバにデータを送信したときに呼び出される。

実際には、Connect()と Control()は使わないが、Ready()と Notify()の使い方だけを覚えれば、オブジェクト間通信のプログラミングには問題はない。
すべてのサブジェクトやオブザーバに対して、オブジェクトはこの 4 つの関数を実装しなければばならないが､次に説明する stub.cfg で null を指定した場合は、stubgen2 コマンドによってデフォルトの Control()、Ready()、Connect()、Notify()が自動的に作成される。

✧ stub.cfg

　stub.cfg は､オブジェクト間通信を簡単に扱えるように必要なファイルを自動生成するために使われる設定ファイル。コンパイル前に stubgen2 コマンドによって補助的なソースファイルを stub.cfg から生成させることで、プログラマは複雑なソースコードを自分で書かずに済む。
以下は SampleSubject と SampleObserver の stub.cfg である。

```
[stub.cfg(SampleSubject)]
ObjectName : SampleSubject
NumOfOSubject : 1
NumOfOObserver : 1
Service : "SampleSubject.SendString.char.S",null,Ready()
Service : "SampleSubject.DummyObserver.DoNotConnect.O",
null, null
[stub.cfg(SampleObserver)]
ObjectName : SampleObserver
NumOfOSubject : 1
NumOfOObserver : 1
Service : "SampleObserver.DummySubject.DoNotConnect.S",
null, null
Service : "SampleObserver.ReceiveString.char.O",null,Notify()
```

＜ObjectName＞
　オブジェクト名を記述。

＜NumOfOSubject＞
　オブジェクトに含まれるサブジェクトの数を記述。

＜NumOfOObserver＞
　オブジェクトに含まれるオブザーバの数を記述。

＜Service(サブジェクトの場合)＞
　ダブルクォーテーションで囲んだサブジェクトのサービス名、Control()の関数名、Ready()の関数名をカンマで区切って指定する。サブジェクトは必ず１つ以上なければならないので、SampleObserver では SampleObserver.

DummySubject .DoNotConnect.S というダミーのサブジェクトを用意している。Control()や Ready()を実装しない場合、関数名の代わりに null を置く。

＜Service(オブザーバの場合)＞
ダブルクォーテーションで囲んだオブザーバのサービス名、Connect()の関数名、Notify()の関数名をカンマで区切って指定する。オブザーバは必ず１つ以上なければならないので、SampleSubject では SampleSubject. DummyObserver .DoNotConnect.O というダミーのオブザーバを用意している。Connect()や Notify()を実装しない場合は、関数名の代わりに null を置く。

オブジェクト間通信を行うので、SampleSubject と SampleObserver の Makefile には、次のような行が追加されている。

```
[Makefile(SampleSubject)]
SampleSubjectStub.cc: stub.cfg
$(STUBGEN) stub.cfg
```

✧ 初期化と終了のためのマクロ

オブジェクト間通信を行うプログラムでは、関数 DoInit()、DoStart()、DoStop()、DoDestroy()にマクロを埋め込んでサブジェクトやオブザーバの準備や後始末を行う。各マクロの役割は次の通り。

＜NEW ALL SUBJECT AND OBSERVER＞
DoInit()で使うマクロで、オブジェクトに含まれるすべてのサブジェクトとオブザーバを生成する。

＜REGISTER ALL ENTRY＞
DoInit()で使うマクロで、サブジェクトの Control()とオブザーバの Connect()をシステムに登録する。

＜SET ALL READY AND NOTIFY ENTRY＞
DoInit()で使うマクロで、サブジェクトの Ready()とオブザーバの Notify()をシステムに登録する。

＜ENABLE ALL SUBJECT＞
DoStart()で使うマクロで、すべてのサブジェクトを使用可能にする。

＜ASSERT READY TO ALL OBSERVER＞
DoStart()で使うマクロで、すべてのオブザーバで準備ができたことを対応するサブジェクトに通知させる。

＜DISABLE ALL SUBJECT＞
DoStop()で使うマクロで、すべてのサブジェクトを使用不可にする。

＜DEASSERT READY TO ALL OBSERVER＞
DoStop()で使うマクロで、すべてのオブザーバがオブジェクト間通信をこれ以上行わないことを対応するサブジェクトに通知させる。

＜DELETE ALL SUBJECT AND OBSERVER＞
DoDestroy()ではこのマクロを使って、オブジェクトに含まれるすべてのサブジェクトとオブザーバを削除する。マクロを扱う場合、各種のマクロを定義している OPENR/core macro.h をインクルードする必要がある。

[例]

```
#include <OPENR/core_macro.h>
#include "MyObject.h"
OStatus
MyObject::DoInit(const OSystemEvent& event)
{
NEW_ALL_SUBJECT_AND_OBSERVER;
REGISTER_ALL_ENTRY;
SET_ALL_READY_AND_NOTIFY_ENTRY;
return oSUCCESS;
}
OStatus
MyObject::DoStart(const OSystemEvent& event)
{
ENABLE_ALL_SUBJECT;
ASSERT_READY_TO_ALL_OBSERVER;
```

```
return oSUCCESS;
}
OStatus
MyObject::DoStop(const OSystemEvent& event)
{
DISABLE_ALL_SUBJECT;
DEASSERT_READY_TO_ALL_OBSERVER;
return oSUCCESS;
}
OStatus
MyObject::DoDestroy(const OSystemEvent& event)
{
DELETE_ALL_SUBJECT_AND_OBSERVER;
return oSUCCESS;
}
```

✧　通信の流れ

SampleSubject と SampleObserver の間で次のようなオブジェクト間通信を行わせる。オブザーバによる要求→サブジェクトによるデータ送信→オブザーバによる受信という流れを繰り返す。

1.SampleObserver::DoStart()

マクロ ASSERT READY TO ALL OBSERVER によって、オブザーバ SampleObserver.ReceiveString.char.O がサブジェクト SampleSubject. SendString.char.S に対してデータを送るように要求する。すると SampleSubject のメンバ関数 Ready()が呼び出される。

2.SampleSubject::Ready()

サブジェクトのメンバ関数 SetData()と NotifyObserver()を使って、オブザーバ Sample Observer.ReceiveString.char.O に文字列を送る。すると SampleObserver のメンバ関数 Notify()が呼び出される。

3.SampleSubject::Ready()

データを受け取って表示する。オブザーバ SampleObserver.

ReceiveString.char.O が AssertReady()を呼ぶことで、再びサブジェクト SampleSubject.SendString.char.S に対してデータを送るように要求する。これによって、SampleSubject::Ready()と SampleObserver::Notify()が繰り返される。

```
[SampleSubject の Ready()]
void
SampleSubject::Ready(const OReadyEvent& event)
{
OSYSPRINT(("SampleSubject::Ready() : %s n",
event.IsAssert() ? "ASSERT READY" :
"DEASSERT READY"));
static int counter = 0;
char str[32];
if (counter == 0) {
//1 回目の送信データ
strcpy(str, "!!! Hello world !!!");
//データをセットする
subject[sbjSendString]->SetData(str,sizeof(str));
  //オブザーバにデータ送信
subject[sbjSendString]->NotifyObservers();
} else if (counter == 1) {
//2 回目の送信データ
strcpy(str, "!!! Hello world again !!!");
  //データをセットする
subject[sbjSendString]->SetData(str,sizeof(str));
  //オブザーバにデータ送信
subject[sbjSendString]->NotifyObservers();
}
counter++;
}
[SampleObserver の Notify()]
void
SampleObserver::Notify(const ONotifyEvent& event)
{
  //データを受信
```

```
const char* text = (const char *)event.Data(0);
  //データを表示
OSYSPRINT(("SampleObserver::Notify() %s  n", text));
  //再びサブジェクトに対して，データ送信を要求
observer[event.ObsIndex()]->AssertReady();
}
```

✧ SampleSubject のソース

[SampleSubject.h]

```
#include <OPENR/OObject.h>
#include <OPENR/OSubject.h>
#include <OPENR/OObserver.h>
#include "def.h"
class SampleSubject : public OObject {
public:
SampleSubject();
virtual ~SampleSubject() {}
OSubject* subject[numOfSubject];
OObserver* observer[numOfObserver];
virtual OStatus DoInit (const OSystemEvent& event);
virtual OStatus DoStart (const OSystemEvent& event);
virtual OStatus DoStop (const OSystemEvent& event);
virtual OStatus DoDestroy(const OSystemEvent& event);
void Ready(const OReadyEvent& event);
};
```

[SampleSubject.cc]

```
#include <string.h>
#include <OPENR/OSyslog.h>
#include <OPENR/core_macro.h>
#include "SampleSubject.h"
SampleSubject::SampleSubject()
{ }
OStatus
SampleSubject::DoInit(const OSystemEvent& event)
```

```
{
NEW_ALL_SUBJECT_AND_OBSERVER;
REGISTER_ALL_ENTRY;
SET_ALL_READY_AND_NOTIFY_ENTRY;
return oSUCCESS;
}
OStatus
SampleSubject::DoStart(const OSystemEvent& event)
{
ENABLE_ALL_SUBJECT;
ASSERT_READY_TO_ALL_OBSERVER;
return oSUCCESS;
}
OStatus
SampleSubject::DoStop(const OSystemEvent& event)
{
DISABLE_ALL_SUBJECT;
DEASSERT_READY_TO_ALL_OBSERVER;
return oSUCCESS;
}
OStatus
SampleSubject::DoDestroy(const OSystemEvent& event)
{
DELETE_ALL_SUBJECT_AND_OBSERVER;
return oSUCCESS;
}
void
SampleSubject::Ready(const OReadyEvent& event)
{
OSYSPRINT(("SampleSubject::Ready() : %s  n",
event.IsAssert() ? "ASSERT READY" :
"DEASSERT READY"));
static int counter = 0;
char str[32];
if (counter == 0) {
strcpy(str, "!!! Hello world !!!");
```

```
subject[sbjSendString]->SetData(str,sizeof(str));
subject[sbjSendString]->NotifyObservers();
} else if (counter == 1) {
strcpy(str, "!!! Hello world again !!!");
subject[sbjSendString]->SetData(str,sizeof(str));
subject[sbjSendString]->NotifyObservers();
}
counter++;
}
```

✧ SampleObserver のソース

[SampleObserver.h]

```
class SampleObserver : public OObject {
public:
SampleObserver();
virtual ~SampleObserver() {}
OSubject* subject[numOfSubject];
OObserver* observer[numOfObserver];
virtual OStatus DoInit (const OSystemEvent& event);
virtual OStatus DoStart (const OSystemEvent& event);
virtual OStatus DoStop (const OSystemEvent& event);
virtual OStatus DoDestroy(const OSystemEvent& event);
void Notify(const ONotifyEvent& event);
};
```

[SampleObserver.cc]

```
#include <OPENR/OSyslog.h>
#include <OPENR/core_macro.h>
#include "SampleObserver.h"
SampleObserver::SampleObserver()
{ }
OStatus
SampleObserver::DoInit(const OSystemEvent& event)
{
NEW_ALL_SUBJECT_AND_OBSERVER;
```

```
REGISTER_ALL_ENTRY;
SET_ALL_READY_AND_NOTIFY_ENTRY;
return oSUCCESS;
}
OStatus
SampleObserver::DoStart(const OSystemEvent& event)
{
ENABLE_ALL_SUBJECT;
ASSERT_READY_TO_ALL_OBSERVER;
return oSUCCESS;
}
OStatus
SampleObserver::DoStop(const OSystemEvent& event)
{
DISABLE_ALL_SUBJECT;
DEASSERT_READY_TO_ALL_OBSERVER;
return oSUCCESS;
}
OStatus
SampleObserver::DoDestroy(const OSystemEvent& event)
{
DELETE_ALL_SUBJECT_AND_OBSERVER;
return oSUCCESS;
}
void
SampleObserver::Notify(const ONotifyEvent& event)
{
const char* text = (const char *)event.Data(0);
OSYSPRINT(("SampleObserver::Notify() %s  n", text));
observer[event.ObsIndex()]->AssertReady();
}
```

5．AIBO を動かす

5.1　OVirtualRobotComm との通信

LED、頭、耳、前脚、後脚などの AIBO の各部を動かすには、プログラマがサブジェクトを含むオブジェクトを作成し、OPEN-R のシステム層に含まれるオブジェクト OVirtualRobotComm のオブザーバに対して命令を出す。前章で SampleSubject から SampleObserver に対して文字列を送ったように、プログラマが作成するサブジェクトを使って、システムを持つオブザーバ OVirtualRobot-Comm.Effector.OCommandVectorData.0 に動作のための情報を含むデータを送信すれば、AIBO を動かすことができる。

OVirtualRobotComm のほかに AIBO を動作させるためのシステムオブジェクトには、AIBO スピーカーから音を鳴らすための OVirtualRobotAudioComm がある。

5.2　OPEN-R API

AIBO に命令を出したり情報を取得したりするには、オブジェクト間通信を使うほかに、場合によっては OPEN-RAPI を呼び出して直接システムとやり取りすることもある。これから説明するサンプルプログラムでは、次の API を使う。

OStatus OPENR::OpenPrimitive(const char* locator、OPrimitiveID* primitiveID)

CPC プリミティブを開く。

OStatus OPENR::NewCommandVectorData(size_t numCommands, MemoryRegionID* memID, OCommandVectorData baseAddr)**

OCommandVectorData を共有メモリ上に作成する。

OStatus OPENR::SetMotorPower(OPower power)

AIBO モーター電源をオンまたはオフにする。
どの API も、成功した場合は定数 oSUCCESS を返す。

5.3 CPC プリミティブ

AIBO は、2 章で前述の通り、コアユニット・頭・脚・尻尾といったユニットで構成されている。これらの部品 CPC(ConfigurablePhysical Component)には、プログラムで操作する対象として、関節・LED・スピーカーといった出力デバイス、カメラや各センサーなどの入力デバイスがあり、これらの操作対象を CPC プリミティブと呼ぶ。

プログラムで CPC プリミティブを指定するための文字列を CPC プリミティブロケータと呼び、ロケータを指定して関数 OPENR::OpenPrimitive()を使って、次のように CPC プリミティブを開くことができる。

```
OPrimitiveID primID;
OStatus result = OPENR::OpenPrimitive(
"PRM:/r1/c1/c2/c3/l1-LED2:l1",&primID);
```

各 CPC プリミティブには、「PRM:/r1/c1/c2/c3/l1 -LED2:l1」のような固有のロケータが割り振られており、第 1 引数にこのロケータを指定する。成功すると第 2 引数には ID が返ってくる。OVirtualRobotComm のオブザーバに送る命令のためのデータを作る時に、この ID を使う。

5.4 命令データの作成

✧ RCRegion

サブジェクトから OVirtualRobotComm のオブザーバに送る命令のデータには、RCRegion という専用のクラスを使う。

```
RCRegion* region;
//中略: region の生成
subject[sbjSubject]->SetData(region);
subject[sbjSubject]->NotifyObservers();
```

RCRegion の「RC」とは、Reference Counter のこと。RCRegion が生成された時点ではカウンタは 1 になっている。OVirtualRobotComm は、データの処理中はカウンタを 2 にする。参照カウンタの数値によって命令データが使用中かどうかわかる。

また、RCRegion は命令データへのポインタを保持する。構造体 OCommandVectorData へのポインタを含む RCRegion を SetData()で送信することで、OVirtualRobotComm のオブザーバに LED を光らせたり、脚を動かしたりする命令を出せる。

✧ OCommandVectorData

OVirtualRobotComm への命令は、構造体 OCommand VectorData で指定する。OCommandVectorData は、関数 OPENR::NewCommandVectorData()を使って共有メモリ上に作成する。第 1 引数は一度に動かす CPC プリミティブの数を指定する。第 3 引数には、作成された OCommandVectorData へのポインタが返る。第 2 引数には、共有メモリに振られた ID が返る。最後に一度に動かす CPC プリミティブの数は、SetNumData()を使って改めて指定しなくてはならない。

OCommandVectorData のポインタを RCRegion に格納する時は、RCRegion のコンストラクタの引数に、vector Info のメンバ関数と OCommandVectorData それ自体のポインタを指定する。

以下の例では、7 つの CPC プリミティブを使うので、第 1 引数を 7 で指定する。

```
RCRegion* region;
MemoryRegionID cmdVecDataID;
OCommandVectorData* cmdVecData;
OPENR::NewCommandVectorData(7,
&cmdVecDataID, &cmdVecData);
region = new RCRegion(
cmdVecData->vectorInfo.memRegionID,
cmdVecData->vectorInfo.offset,
(void*)cmdVecData,
cmdVecData->vectorInfo.totalSize);
cmdVecData->SetNumData(7);
```

OCommandVectorData は、次のように定義されている。ODataVectorInfo は、OCommandVectorData 全体に関する情報を含む。OComm andInfo は、OCommand VectorData に格納された複数の OCommandData のそれぞれに対する情報を含む。

```
struct OCommandVectorData {
```

```
ODataVectorInfo vectorInfo;
OCommandInfo info[1];
void SetNumData(size_t ndata) {
    vectorInfo.numData = ndata;
  }
OCommandInfo* GetInfo(int index) {
    return &info[index];
  }
OCommandData* GetData(int index) {
return (OCommandData*)((byte*)&vectorInfo +
          info[index].dataOffset);
}
};
```

メモリ上の OCommandVectorData から OCommandInfo や OCommandData を取り出すには、GetInfo()や Get Data()に配列のインデックス番号を指定する。

```
for(int i=0; i<numOfData; i++){
OCommandInfo* info = cmdVecData->GetInfo();
OCommandData* data = cmdVecData->GetData();
}
```

✧ 命令データの作成

OCommandVectorData を作成したら、その中の OCommandInfo と OCommandData の配列要素のそれぞれにデータを入れることで、命令データが出来上がる。

＜OCommandInfo＞

まず、OCommandInfo に関数 Set()でデータを指定する。Set()の第１引数は動かす CPC プリミティブによって変わる定数。第２引数は OPENR::OpenPrimitive()で開いた CPC プリミティブの ID。第３引数はフレーム数、つまり OCommandData 内の配列要素。

LED の場合

```
OCommandInfo* info = cmdVecData->GetInfo(i);
info->Set(odataLED_COMMAND2, primID, 1);
```

関節の場合

```
OCommandInfo* info = cmdVecData->GetInfo(i);
info->Set(odataJOINT_COMMAND2, primID,
ocommandMAX_FRAMES);
```

<OCommandData>

次に OCommandData にデータを入れる。OCommandData は次のように定義される。OCommandValue は 8 バイトの型で、定数 ocommandMAX FRAMES は 16 なので、OCommandData には 128 バイトまでのデータを収められる。

```
struct OCommandData{
OCommandValue value[ocommandMAX_FRAMES];
}
```

実際には、OCommandValue をそのまま使わず、OCommandValue の配列 value を CPC プリミティブの種類に合わせて使う。

LED の場合

```
OCommandData* data = cmdVecData->GetData(i);
OLEDCommandValue2* val =
(OLEDCommandValue2*)data->value;
```

関節の場合

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

✧ まとめ

OVirtualRobotComm へ送る命令データの作り方は次の手順。

1. OPENR::NewCommandVectorData()で構造体 OCommandVectorData を生成。1 つの OCommand VectorData の中には複数のデータ（OCommandInfo と OCommandData）を保持できる。
2. OCommandVectorData へのポインタを RCRegion に格納する。
3. OCommandInfo と OCommandData の配列要素に命令データを入れる。
4. RCRegion を引数にして SetData()を呼び出し、OVirtualRobotComm のオブザーバに命令を送る。

６．実際のプログラム例

今回は、MovingLegs のプログラムの詳細を載せる。

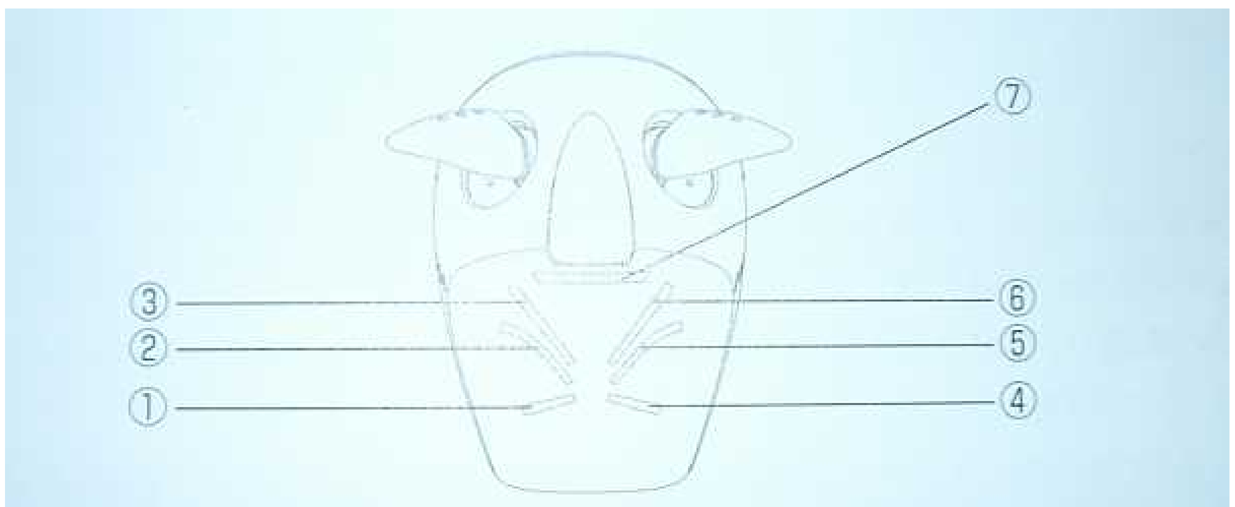
起動すると 4 本の脚を動かして伏せのポーズをとり、そのまま静止。耳と頭を左右に動かしながら、ＬＥＤを光らせる。実際のプログラムは別途添付するものとする。以下はプログラム内容の補足説明である。

[オブジェクト]

- BlinkingLED　　ＬＥＤ点滅
- MovingEar　　耳を動かす
- MovingHead　　首を左右に動かす
- MovingLegs　　脚を動かして伏せのポーズにする
- PowerMonitor　　ポーズスイッチ、バッテリーの監視

＜BlinkingLED＞

各関節のロケータは次の通り。



①目ランプ（左下）・・・PRM:/r1/c1/c2/c3/l1-LED2:l1
②目ランプ（左中）・・・PRM:/r1/c1/c2/c3/l2-LED2:l2
③目ランプ（左上）・・・PRM:/r1/c1/c2/c3/l3-LED2:l3

④目ランプ（右下）・・・PRM:/r1/c1/c2/c3/l4-LED2:l4
⑤目ランプ（右中）・・・PRM:/r1/c1/c2/c3/l5-LED2:l5
⑥目ランプ（右上）・・・PRM:/r1/c1/c2/c3/l6-LED2:l6
⑦モードランプ ・・・・PRM:/r1/c1/c2/c3/l7-LED2:l7

＜MovingEar＞

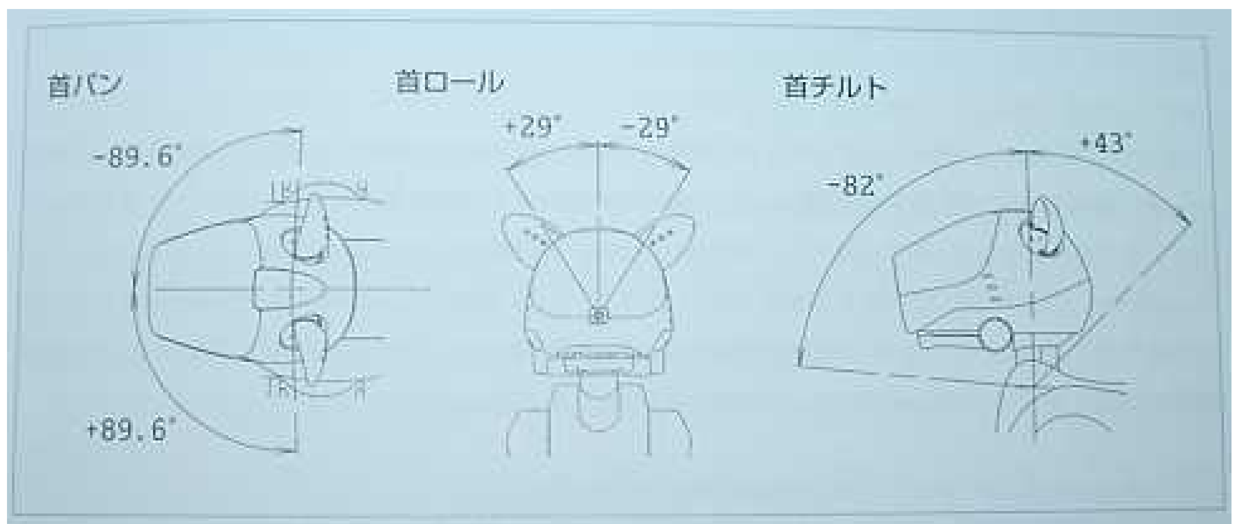
各関節のロケータは次の通り。

左耳・・・PRM:/r1/c1/c2/c3/e1-Joint3:j5
右耳・・・PRM:/r1/c1/c2/c3/e2-Joint3:j6

BlinkingLED とMovingEar はともに、OCommandVectirData を作成し、その中の OCommandInfo と OCommandData に情報をセットする。

＜MovingHead＞

各関節のロケータは次の通り。



首チルト・・・PRM:/r1/c1-joint2:j1
首パン・・・PRM:/r1/c1/c2-joint2:j2
首ロール・・・PRM:/r1/c1/c2/c3-joint2:j3

＜MHS_IDLE＞

関数 DoStart()が呼ばれる前の状態は MLS_IDLE。また、関数 DoStop()が呼ばれると MLS_IDLE に戻す。

＜MHS_IDLE → MHS__ADJUTING_DIFF_JOINT_VALUE＞

関数 DoStart()では、OVisualRobotComm のオブザーバの準備ができている場合、関節の角度を読み取って、現在地をそのまま指示値として送信する。そして、状態を MHS_ADJUTING_DIFF_JOINT_VALUE とする。

＜MHS_IDLE → MHS_START＞

関数 DoStart()では、OVisualRobotComm のオブザーバの準備ができていない場合、何もせずに状態を MHS_START とする。

＜MHS_START → MHS_ADJUTING_DIFF_JOINT_VALUE＞

関数 DoReady では、状態が MHS_START の場合関節の角度を読み取って、現在地をそのまま指示値として送信する。そして、状態を MHS_ADJUTING_DIFF_JOINT_VALUE とする。

＜MHS_ADJUTING_DIFF_JOINT_VALUE→MHS_MOVING_TO_ZERO_POS＞

関数 DoReady では、状態が MHS_ADJUTING_DIFF_JOINT_VALUE の場合、ゲインをセットし、首を原点に動かしはじめる。そして、その状態を MHS_MOVING_TO_ZERO_POS とする。

＜MHS_MOVING_TO_ ZERO_POS → MLS_SWING_HEAD＞

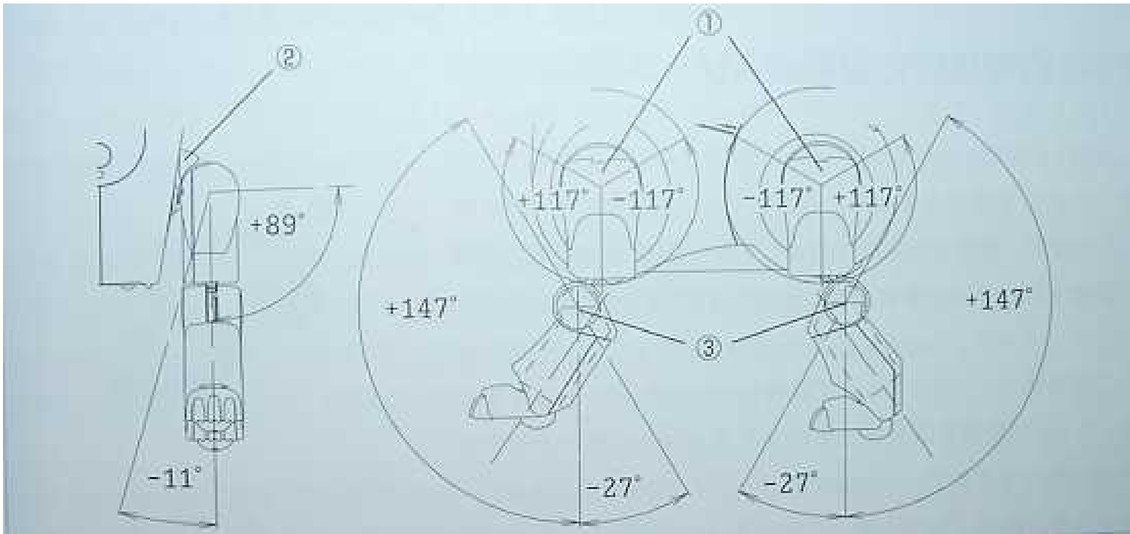
関数 DoReady では、状態が MHS_ADJUTING_DIFF_JOINT_VALUE の場合、首を原点に動かし続ける。首が原点に到達したら、状態を MHS_SWING_HEAD とする。

＜MHS_SWING_HEAD＞

関数 DoReady では、状態が **MHS_SWING_HEAD** の場合、首を左右に振り続ける。

＜MovingLegs＞

各関節のロケータは次の通り。



４本の脚にそれぞれ３つずつついているＣＰＣプリミティブ、計 12 個の関節を動かす。

右前脚・・・①PRM:/r4/c1-joint2:j1
②PRM:/r4/c1/c2-joint2:j2
③PRM:/r4/c1/c2-/c3-joint2:j3

左前脚・・・①PRM:/r2/c1-joint2:j1
②PRM:/r2/c1/c2-joint2:j2
③PRM:/r2/c1/c2-/c3-joint2:j3

右後脚・・・①PRM:/r5/c1-joint2:j1
②PRM:/r5/c1/c2-joint2:j2
③PRM:/r5/c1/c2-/c3-joint2:j3

左後脚・・・①PRM:/r3/c1-joint2:j1
②PRM:/r3/c1/c2-joint2:j2
③PRM:/r3/c1/c2-/c3-joint2:j3

＜MLS_IDLE＞

関数 DoStart()が呼ばれる前の状態は MLS_IDLE。また、関数 DoStop()が呼ばれると MLS_IDLE に戻す。

＜MLS_IDLE → MLS_ADJUTING_DIFF_JOINT_VALUE＞

関数 DoStart()では、OVisualRobotComm のオブザーバの準備ができている場合、関節の角度を読み取って、現在地をそのまま指示値として送信する。そして、状態を MLS_ADJUTING_DIFF_JOINT_VALUE とする。

＜MLS_IDLE → MLS_START＞

関数 DoStart()では、OVisualRobotComm のオブザーバの準備ができていない場合、何もせずに状態を MLS_START とする。

＜MLS_START → MLS_ADJUTING_DIFF_JOINT_VALUE＞

関数 DoReady では、状態が MLS_START の場合関節の角度を読み取って、現在地をそのまま指示値として送信する。そして、状態を MLS_ADJUTING_DIFF_JOINT_VALUE とする。

＜MLS_ADJUTING_DIFF_JOINT_VALUE→MLS_MOVING_TO_BROADBASE＞

関数 DoReady では、状態が MLS_ADJUTING_DIFF_JOINT_VALUE の場合、ゲインをセットし、脚を広げた姿勢に移行いはじめる。そして、その状態を MLS_MOVING_TO_BROADBASE とする。

＜MLS_MOVING_TO_BROADBASE → MLS_MOVING_TO_SLEEPING＞

関数 DoReady では、状態が MLS_ADJUTING_DIFF_JOINT_VALUE の場合、脚を広げた姿勢に移行するように関節を動かし続ける。脚を広げた状態が完了したら、状態を MLS_MOVING_TO_SLEEPING とする。

＜MLS_MOVING_TO_SLEEPING＞

関数 DoReady では、状態が MLS_MOVING_TO_SLEEPING の場合、スリープ姿勢に移行するように関節を動かし続ける。スリープ姿勢が完了したら、状態を MLS_IDLE に戻す。

7．おわりに

＜反省点＞

AIBO のプログラミング方法を理解するまでに時間がかかってしまい、実際にプログラム作りに手をつけた時期が遅くなってしまった。

実際に作り始めると、サンプルでも動かなかったり、理解できないところがあるたびにつまずいてしまったりした。そのため、結局目標としていた大きなプログラム作りまで手が届かず、サンプルプログラムを改造したり、本当に単純な動作をさせたりというだけにとどまってしまったことがくやしい。

今回は、山本昌弘教授ゼミ所有の AIBO をお借りしていたため、卒業するにあたり、返還して私の手元にはなくなってしまうため、これ以上の作品を作れなくなってしまう事は残念だ。

＜感想＞

プログラムの内容に関しては、反省点としてあげたとおりであり、大変くやしさが残る。しかし、AIBO のプログラミング方法を一から学び、私がゼロから書いたプログラムで少しでも動いてくれた感動は忘れない。

今回の目的の一つであった、「きちんと計画を立てながら目標とするプログラムを作る」という事は、残念ながら達成できたとはいえないが、大前提であった「C＋+言語の習得」は達成できた。授業で学んだ基礎を思い出しつつ、2 冊の本を読み込み、C＋＋言語を用いたプログラムはある程度かけるようになったと思う。

また、ゼミを受講したこの 2 年間で、メリハリを付けて集中する力と、物事を簡潔にわかりやすく数値化して考えるという、プログラミング独特の思考力が身につけられた。この力は、プログラミングをする際だけではなく、普段の生活やこれからの仕事などにおいても、十分発揮できる能力だと思うので、学んだ事を忘れずに今後もぜひ生かしていきたいと思う。

＜謝辞＞

1 年間改めてプログラミングに対する考え方の基礎を教えてくださり、数々のアドバイスをくださった重定先生、プログラミングの楽しさを最初に吹き込んでくださり、人間的にもたくさんのことを教えてくださった山本昌弘先生に深く感謝申し上げます。また、この 2 年間のゼミ生活の中で、私を支えてくださった先輩後輩、同期の仲間たちにも感謝します。

～参考文献・引用～

- C++で AIBO を自在に動かす　OPEN-R プログラミング入門
  著者：OPEN-R プログラミング SIG
  発行人：土田米一
  編集人：辻本英二
  発行：株式会社インプレス

- 学生のための C
  著者：内山章夫　　河野吉伸
  　　　津村栄一　　中村隆一
  　　　長谷川洋介
  発行所：東京電機大学出版局

- 独習 C
  著者：Herbert Schildt
  翻訳：トップスタジオ
  監修：柏原正三
  出版社：翔泳社

- 独習 C++
  著者：Herbert Schildt
  翻訳：トップスタジオ
  監修：神林靖
  出版社：翔泳社

～参考 URL～

- OPEN-R オフィシャルウェブサイト→http://www.aibo.com/openr/
- AIBO オフィシャルホームページ→http://www.jp.aibo.com