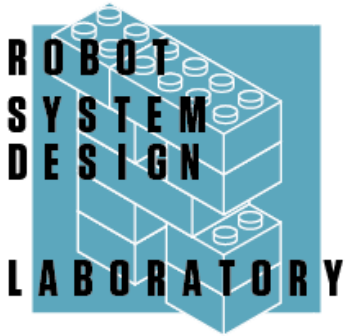
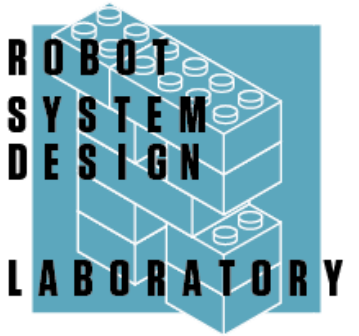


第3部
RTシステム構築演習

第3部の目的

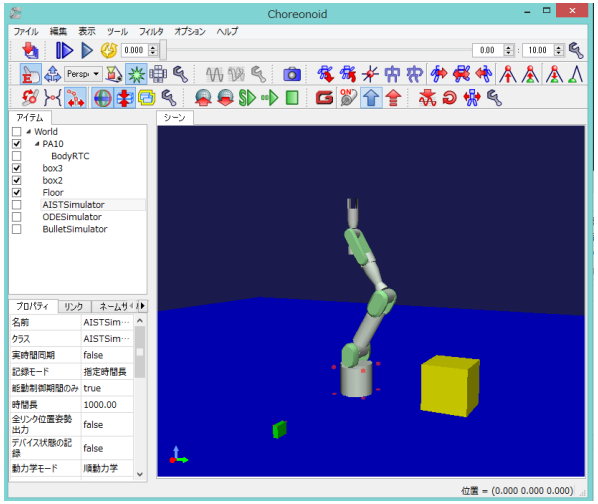
- ロボットアームを用いたRTC群の動作確認と，実際の開発を通じて，RTコンポーネントの開発プロセスを体験する.
- サービスポートを用いたRTコンポーネントの作成方法を体験する.

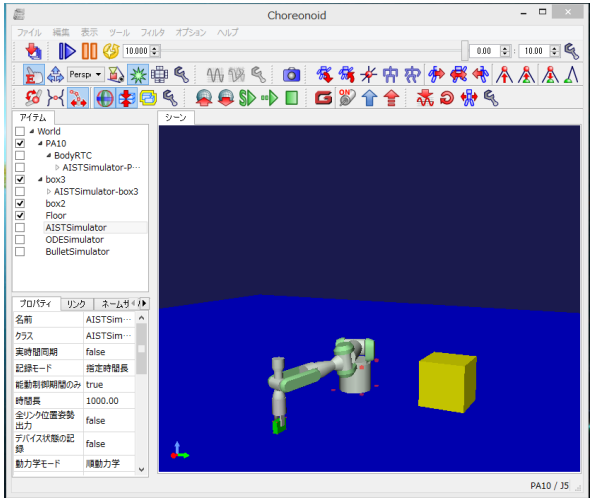


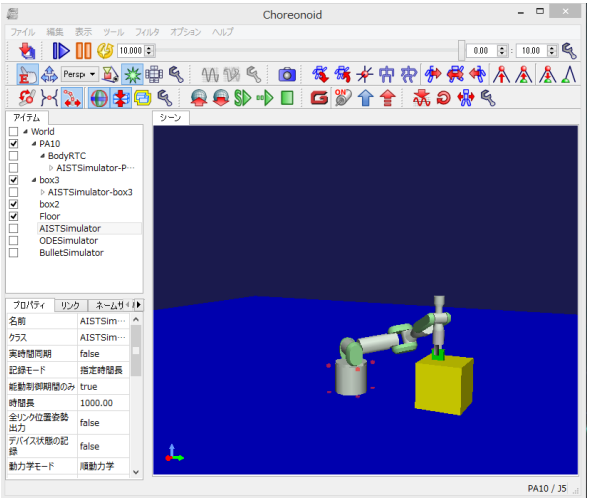
ロボットアームRTC群を用いた RTミドルウェアのロボット応用体験

Choreonoidを用いたアームRTC群の動作体験

- ロボットシミュレータChoreonoidのPA10を操作するRTCの動作を体験する.
- ChoreonoidとPA10のモデル，制御用RTCについては，配布したものを利用
- 本システムでは，手先の位置と姿勢を入力してPA10で緑の箱を移動させる.







Choreonoidとは？

- 産業技術総合研究所で開発されているロボット用統合GUIソフトウェア.
- 動力学シミュレーション，動作振り付け機能を標準装備.
- 独自の機能もプラグインとして追加可能.

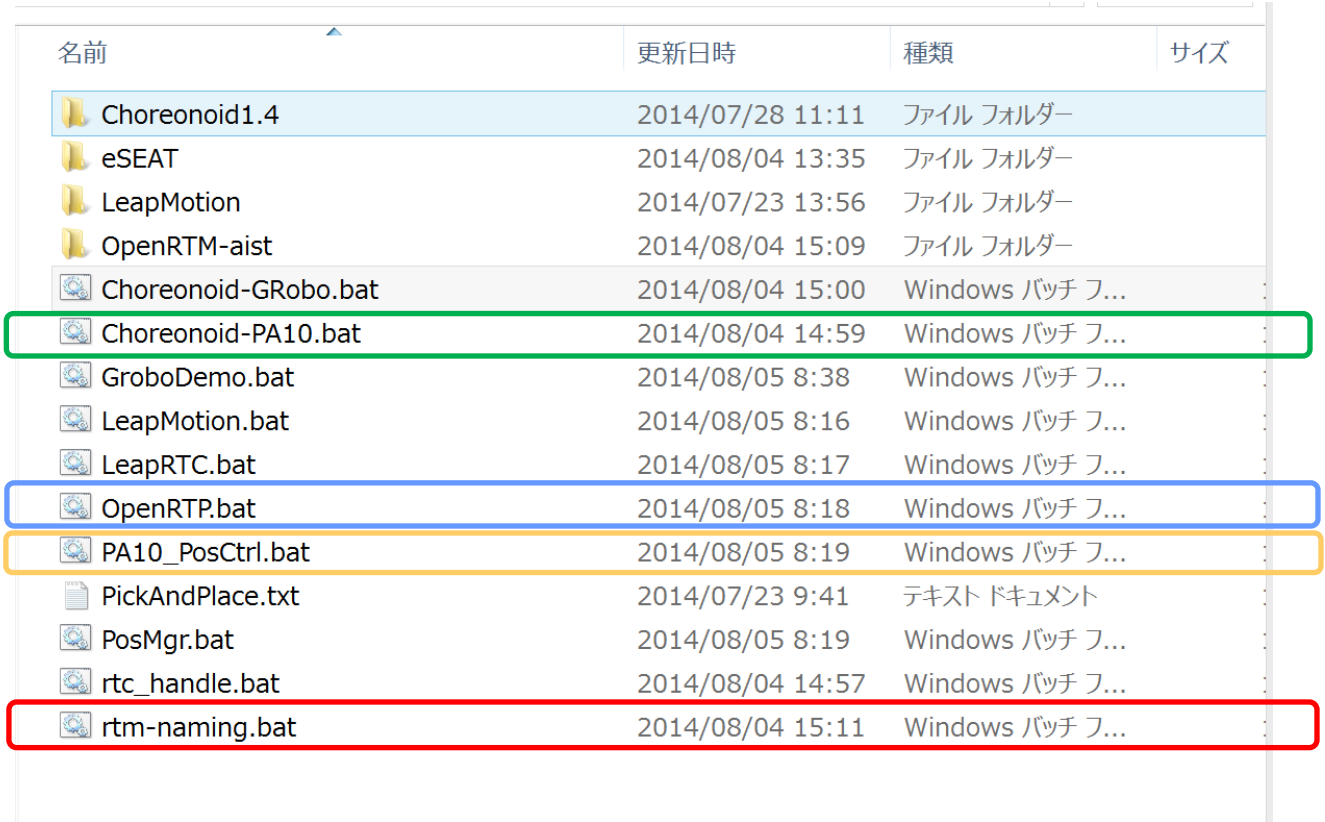
Choreonoid HPより引用

Choreonoid HP

http://choreonoid.org/ja/

Choreonoidを用いたアームRTC群の動作体験

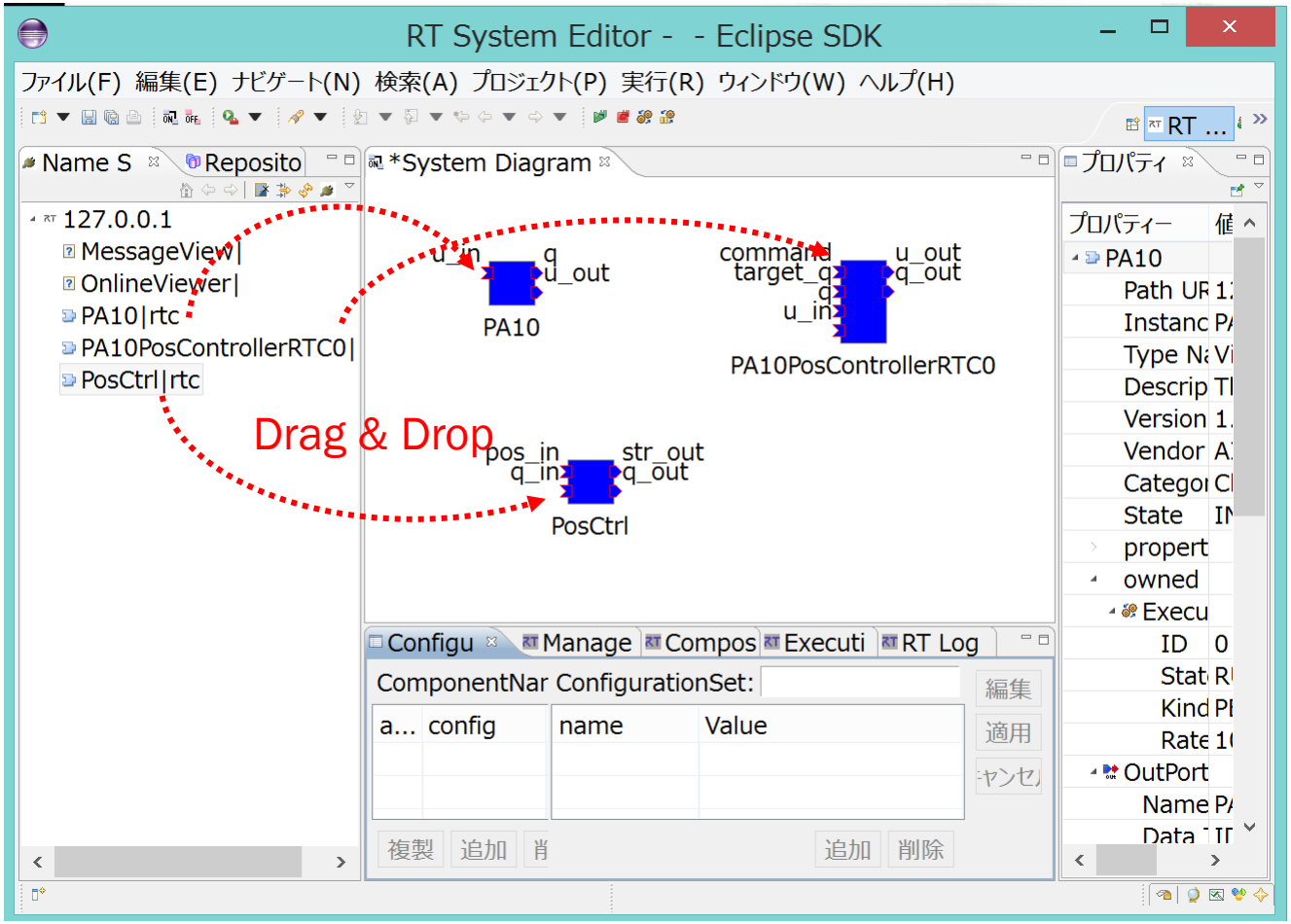
- NameServerの起動： rtm-naming.bat
- RT SystemEditorの起動：OpenRTP.bat
- Choreonoidの起動：Chorenoid-PA10.bat
- PA10のコントローラの起動： PA10_PosCtrl.bat



名前	更新日時	種類	サイズ
Choreonoid1.4	2014/07/28 11:11	ファイル フォルダー	
eSEAT	2014/08/04 13:35	ファイル フォルダー	
LeapMotion	2014/07/23 13:56	ファイル フォルダー	
OpenRTM-aist	2014/08/04 15:09	ファイル フォルダー	
Choreonoid-GRobo.bat	2014/08/04 15:00	Windows バッチ フ...	
Choreonoid-PA10.bat	2014/08/04 14:59	Windows バッチ フ...	
GroboDemo.bat	2014/08/05 8:38	Windows バッチ フ...	
LeapMotion.bat	2014/08/05 8:16	Windows バッチ フ...	
LeapRTC.bat	2014/08/05 8:17	Windows バッチ フ...	
OpenRTP.bat	2014/08/05 8:18	Windows バッチ フ...	
PA10_PosCtrl.bat	2014/08/05 8:19	Windows バッチ フ...	
PickAndPlace.txt	2014/07/23 9:41	テキスト ドキュメント	
PosMgr.bat	2014/08/05 8:19	Windows バッチ フ...	
rtc_handle.bat	2014/08/04 14:57	Windows バッチ フ...	
rtm-naming.bat	2014/08/04 15:11	Windows バッチ フ...	

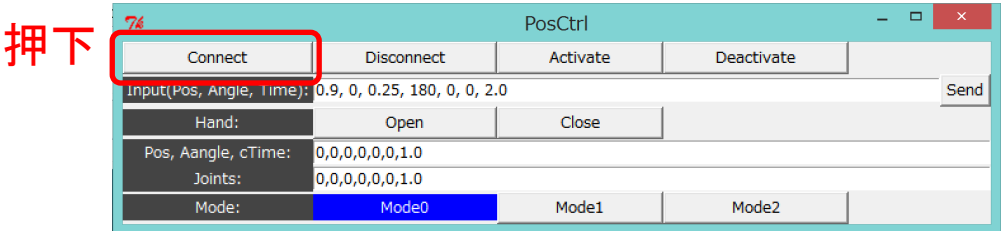
Choreonoidを用いたアームRTC群の動作体験

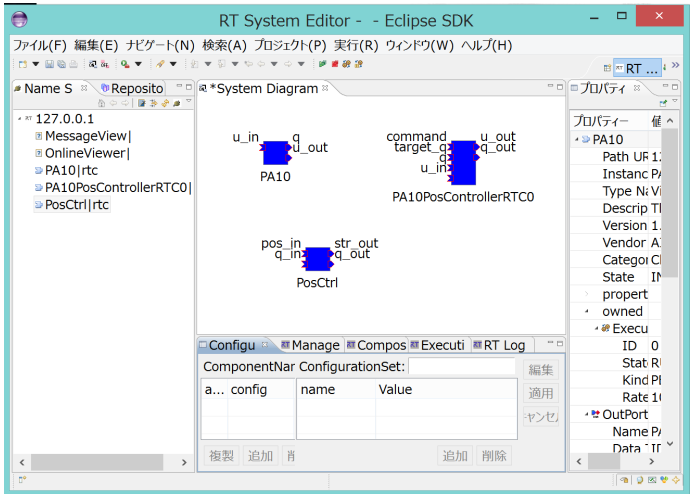
- RT System Editorを操作して，コンポーネントを表示する.



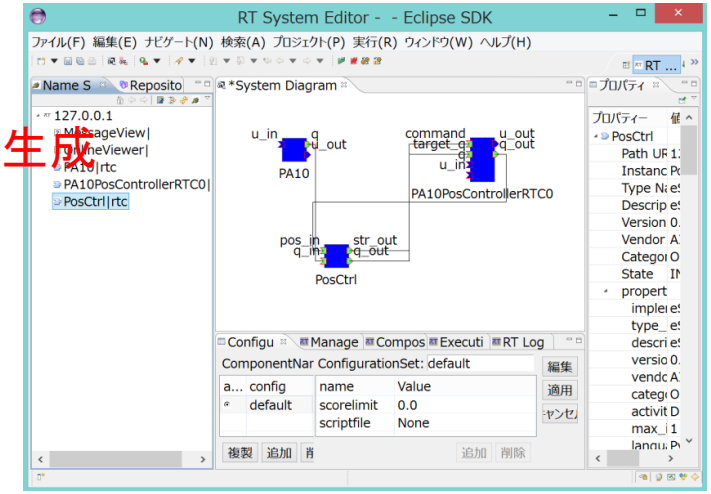
Choreonoidを用いたアームRTC群の動作体験

- PA10_PosCtrlの上部のボタンを操作して、コンポーネント間の接続を生成し、RT　System Editorで確認する



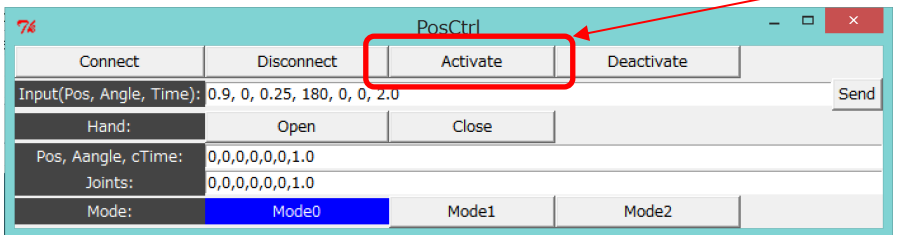


コネクションの生成

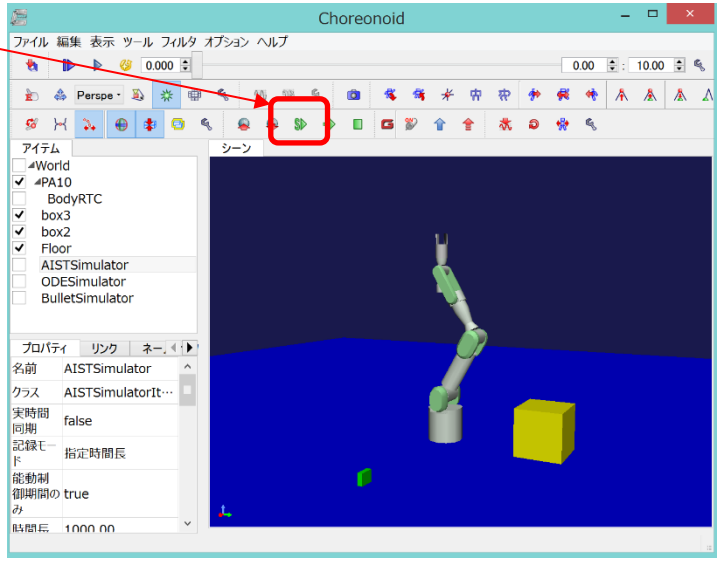


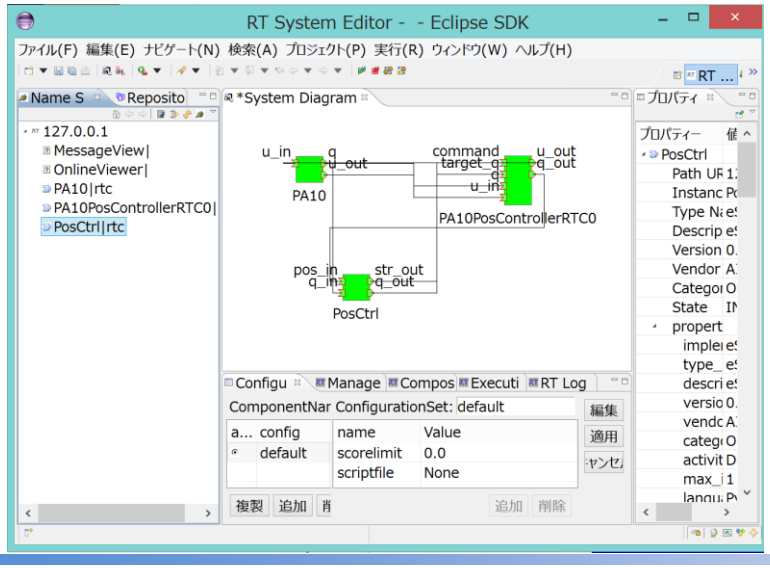
Choreonoidを用いたアームRTC群の動作体験

- PA10_PosCtrlの上部のボタンを操作して、コンポーネントを有効化し、Choreonoidのシミュレーションを開始する



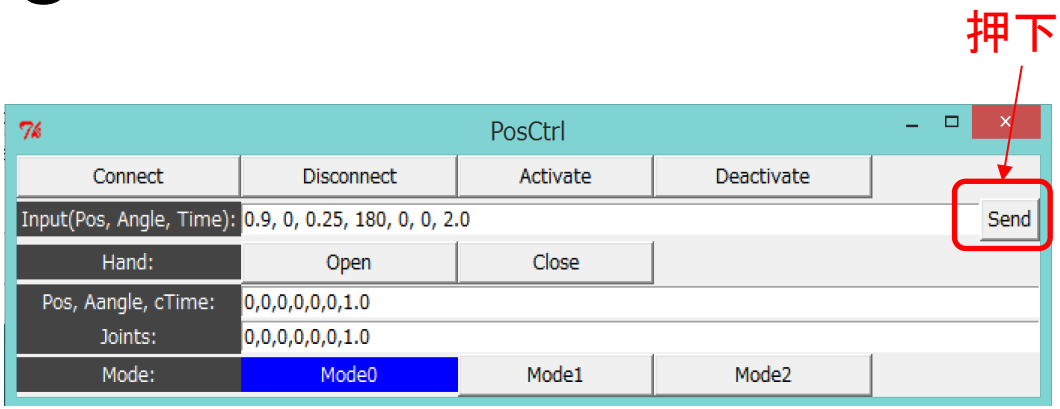
押下

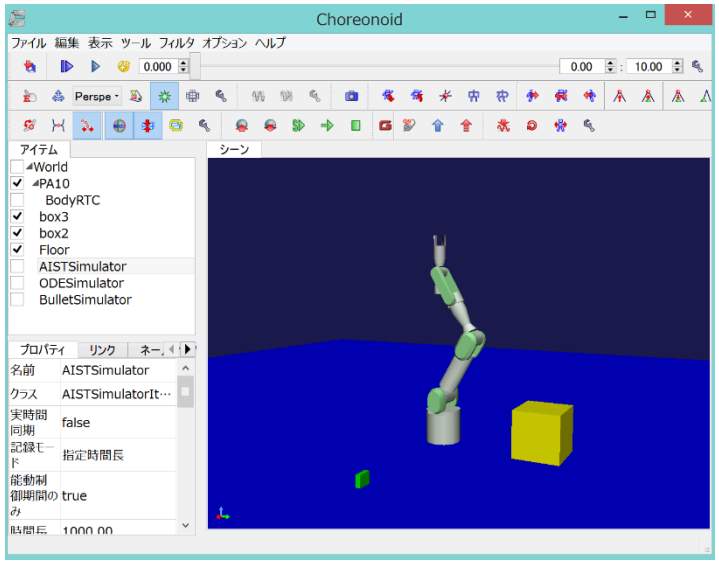




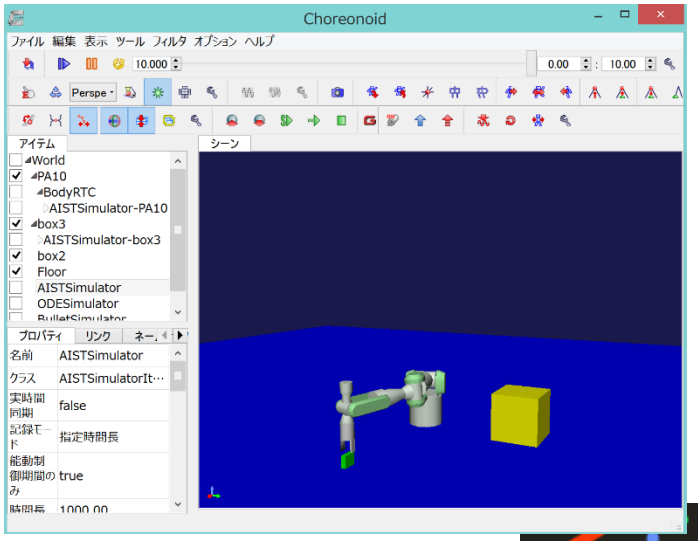
Choreonoidを用いたアームRTC群の動作体験

- PA10_PosCtrlを操作して、Choreonoid内のPA10を操作する.



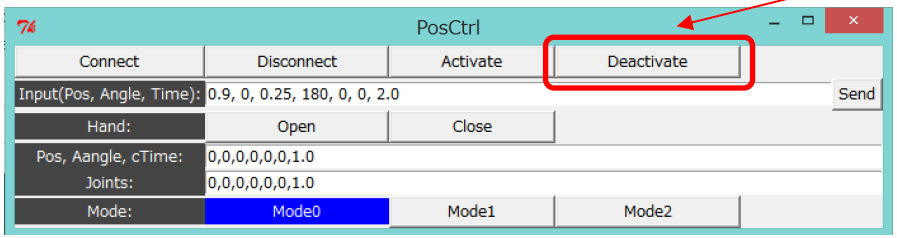


PA10の手先を
目標位置へ移動

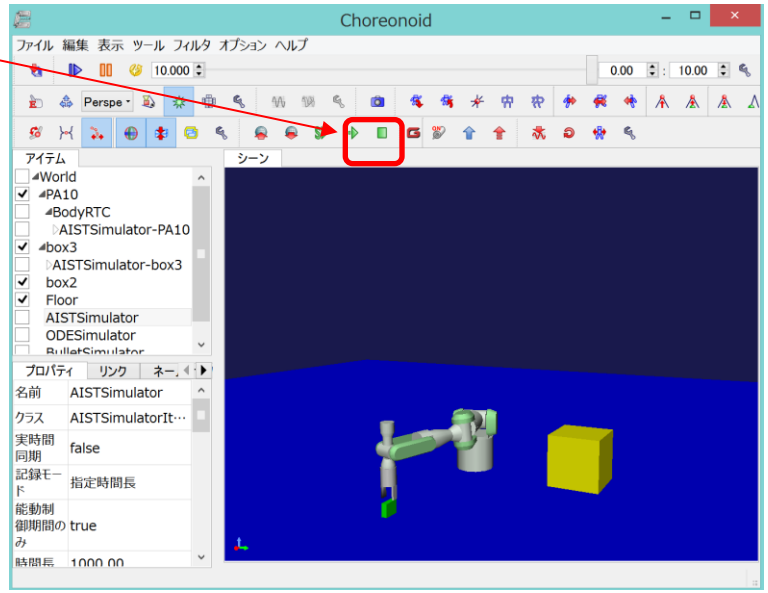


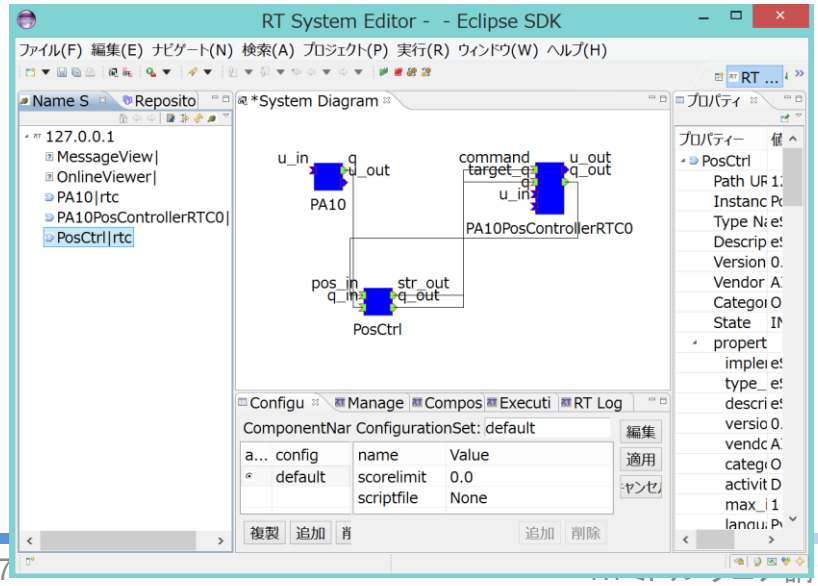
Choreonoidを用いたアームRTC群の動作体験

- PA10_PosCtrlの上部のボタンを操作して、コンポーネントを無有効化し、Choreonoidのシミュレーションを終了する



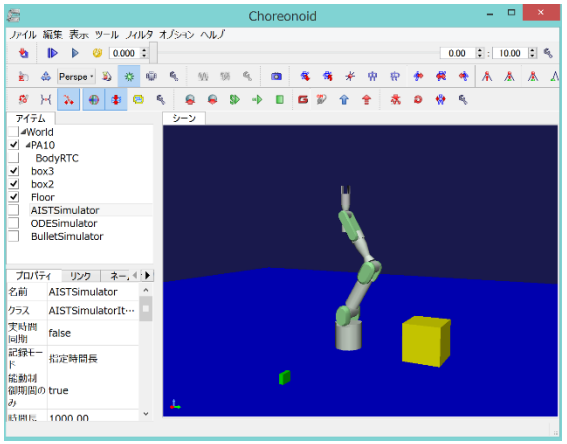
押下

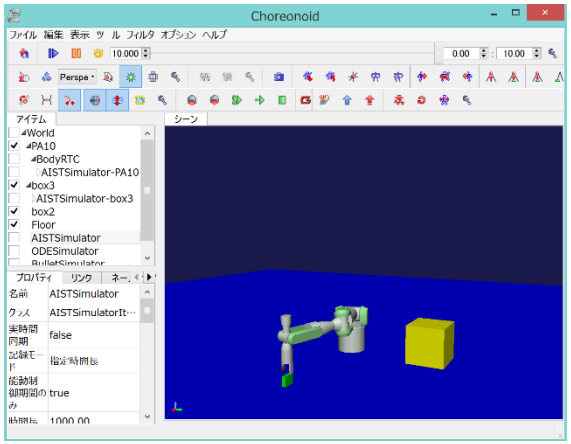


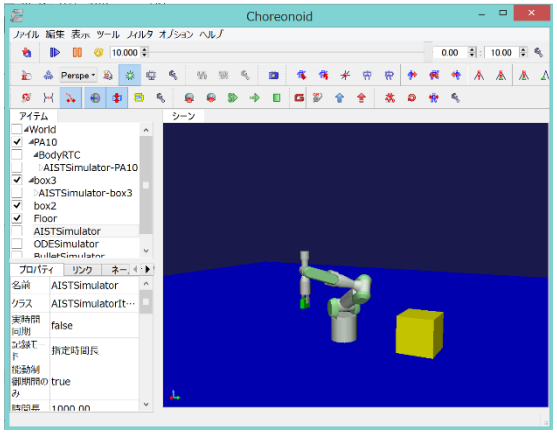


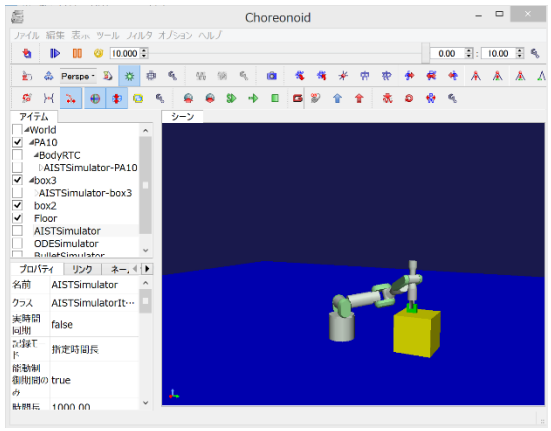
RTコンポーネント開発演習

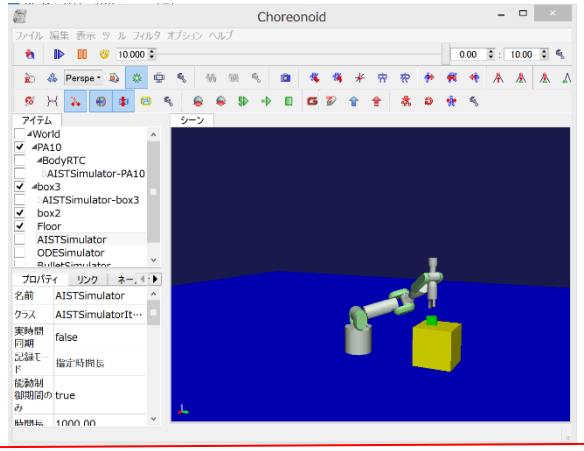
- PA10_PosCtrlと同じようにChoreonoid内のPA10を制御して，緑色の箱を移動させる

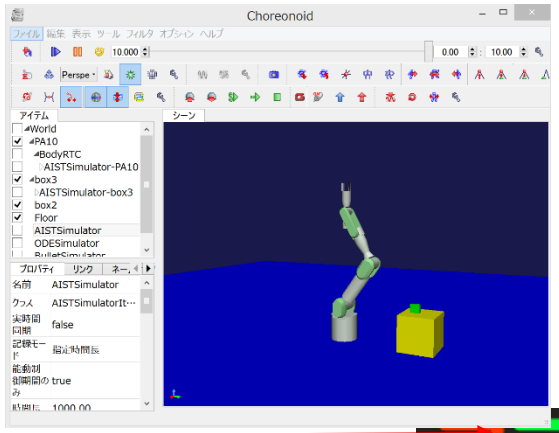












RTコンポーネント開発演習

- PA10_PosCtrlと同じようにChoreonoid内のPA10を制御して、緑色の箱を移動させるRTCをVC++で実装する。
- 各動作の目標位置は、(X, Y, Z, roll, pitch, yaw)で与える

初期姿勢:0.0, 0.025, 1.2, 0.0, 0, 0, 2.0
アプローチ点:0.9, 0.0, 0.25, 180, 0, 0, 2.0
把持点:0.9, 0.0, 0.20, 180, 0, 0, 2.0
移動中間点:0.6, 0.0, 0.60, 180, 0, 0, 2.0
移動先アプローチ点:0.0, 0.65, 0.6, 180, 0, 90, 2.0
移動先:0.0, 0.65, 0.5, 180, 0, 90, 2.0
退避点:0.0, 0.65, 0.6, 180, 0, 90, 2.0
最終姿勢:0.0, 0.025, 1.2, 0.0, 0.0, 0.0, 2.0

- 把持点でハンドを閉じ、移送先でハンドを開けるようにする
- 各移動終了がわかるように、現在のロボットの位置姿勢を読み込むようにする.

RTコンポーネントの動作フロー

- RTCの生成時　→　onInitialize　を実行し待機状態に移行
- RTCの有効化　→　onActivated を実行し実行状態に移行

  （RT SystemEditorまたはrtshellなどから操作）
- RTCの実行時　→　onExecuteを周期実行
  - 実行周期は、rtc.confで指定可能
- RTCの無効化　→　onDeactivatedを実行し待機状態に移行

  （RT SystemEditorまたはrtshellなどから操作）

コアロジックの実装

0. PA10の目標位置姿勢のデータ、ローカル変数を追記
1. PA10Sampleがアクティベートされたときに、初期姿勢になるように目標位置姿勢をPA10PosContollerRTCに送信
2. ロボットハンドが動作し始めると現在の位置姿勢が送信されてきますので、与えた目標位置姿勢と現在の位置姿勢の差分dを計算
3. dのノルムが十分に小さい場合、目標位置姿勢に到達したと考えられるので、次の目標位置姿勢をPA10PosControllerRTCに送信
4. 以降2,3を繰り返す

コアロジックの実装

0. PA10の目標位置姿勢のデータ、ローカル変数を追記

```
static const double path[]={
        0.0, 0.025, 1.2, 0, 0, 0, 2.0,
        0.9, 0.0, 0.25, 180, 0, 0,2.0,
        0.9, 0.0, 0.20, 180, 0, 0, 2.0,
        0.6, 0.0, 0.60, 180, 0, 0, 2.0,
        0.0, 0.65, 0.6, 180, 0, 90, 2.0,
        0.0, 0.65, 0.5, 180, 0, 90, 2.0,
        0.0, 0.65, 0.6, 180, 0, 90, 2.0,
        0.0, 0.025, 1.2, 0.0, 0.0, 0.0, 2.0
};
```

X=0.9m, Y=0.0m, Z=0.25m
r=180°, p=0°, y=0°
Time（移動時間）=2.0秒

PA10Sample.cpp

```
int nPath, count;
double *targetPos;
```

目標位置姿勢のインデックス

目標位置姿勢の状態のカウント

目標位置姿勢

PA10Sample.h

コアロジックの実装

0. PA10の目標位置姿勢のデータ、ローカル変数を追記

```
PA10Sample::PA10Sample(RTC::Manager* manager)
    // <rtc-template block="initializer">
  : RTC::DataFlowComponentBase(manager),
    m_Pos_inIn("Pos_in", m_Pos_in),
    m_CommandOut("Command", m_Command),
    m_Pos_outOut("Pos_out", m_Pos_out)

    // </rtc-template>
{
    nPath = 0;
    count = 0;
    targetPos = new double[7];
}
```

入出力ポートの内部変数の初期化

追記した内部変数の初期化

コアロジックの実装

1. PA10Sampleがアクティベートされたときに、初期姿勢になるように目標位置姿勢をPA10PosContollerRTCに送信

```
RTC::ReturnCode_t PA10Sample::onActivated(RTC::UniqueId ec_id)
{
  nPath=count=0;
  m_Pos_out.data.length(7);
```

出力データポートのためのバッファを確保

```
  for(int i=0; i<7;i++){
    m_Pos_out.data[i] = targetPos[i] = path[nPath*7+i];
  }
```

初期姿勢を出力データに代入

```
  m_Pos_outOut.write();
```

データを出力ポートへ書き込み。
送信は、自動的に行う。

```
  return RTC::RTC_OK;
}
```

コアロジックの実装

2. ロボットハンドが動作し始めると現在の位置姿勢が送信されてきますので、与えた目標位置姿勢と現在の位置姿勢の差分dを計算

```
RTC::ReturnCode_t PA10Sample::onExecute(RTC::UniqueId ec_id)
{
  double diffVal=0;

  if(m_Pos_inIn.isNew()){
    m_Pos_inIn.read();

    for(int i=0; i < 6;i++){
      diffVal += fabs(targetPos[i] - m_Pos_in.data[i]);
    }

...
  }
  return RTC::RTC_OK;
}
```

入力データポートをチェック

入力データポートからデータを読み込み

与えた目標位置姿勢と現在の位置姿勢の差分を計算

コアロジックの実装

3. dのノルムが十分に小さい場合、目標位置姿勢に到達したと考えられるので、次の目標位置姿勢をPA10PosControllerRTCに送信

```
RTC::ReturnCode_t PA10Sample::onExecute(RTC::UniqueId ec_id)
{

...
  if (diffVal < 0.0001){ count++; }
  else { count = 0; }

    if (count > 0 && nPath < 8){
      nPath++;
      for(int i=0; i<7;i++){
        m_Pos_out.data[i] = targetPos[i] = path[nPath*7+i];
      }
      m_Pos_outOut.write();
    }
  }
  return RTC::RTC_OK;
}
```

与えた目標位置姿勢と現在の位置姿勢の差分が十分小さいことを検証

次の目標位置姿勢を出力

コアロジックの実装

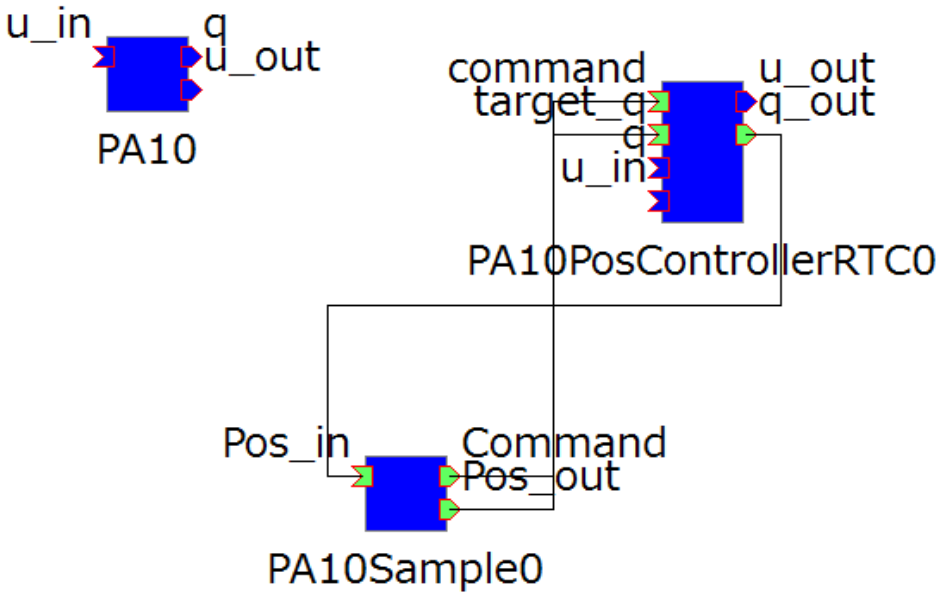
```
RTC::ReturnCode_t PA10Sample::onExecute(RTC::UniqueId ec_id)
{
  double diffVal=0;

  if(m_Pos_inIn.isNew()){
      m_Pos_inIn.read();
      for(int i=0; i < 6;i++){   diffVal += fabs(targetPos[i] - m_Pos_in.data[i]);   }
      if (diffVal < 0.0001){ count++; }   else { count = 0; }
      if (count > 0 && nPath < 8){
          if (nPath == 2 && count < 1000){
              if (count == 1){
                  m_Command.data="Close";
                  m_CommandOut.write();
              }
          }else if(nPath == 5 &&  count < 1000){
              if (count == 1){
                  m_Command.data="Open";
             m_CommandOut.write();
              }
          }else{
              nPath++;
              for(int i=0; i<7;i++){   m_Pos_out.data[i] = targetPos[i] = path[nPath*7+i];   }
              m_Pos_outOut.write();
          }
      }
  }
  return RTC::RTC_OK;
}
```

ハンドを閉じる命令を送信し、1000ループ待つ

ハンドを開く命令を送信し、1000ループ待つ

コンパイルと実行

- VC++2010のソリューションを開きビルド、エラーがなければ実行
- Choreonoidを起動（Choreonoid-PA10.bat）
- RT SystemEditorで下記のように接続



- Choreonoidのシミュレーションを開始し、PA10Sample0をRT SystemEditorで実行状態にする
- PA10の動作を確認

応用編(サービスポートの使い方)

サービスポートの利点

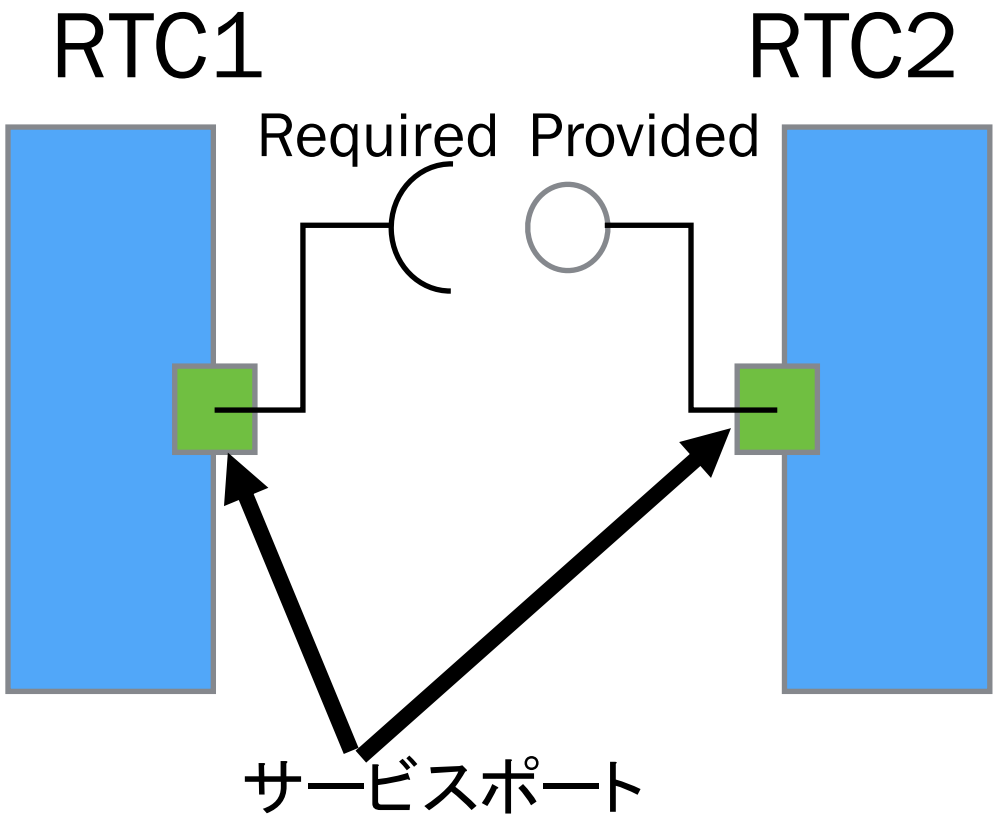
- RTコンポーネントに関数で呼び出せるインタフェースを追加.
  - 引数でデータを渡すことや，返値でデータやRTコンポーネントの状態を受け取ることができる.

データポートとの比較

- データの要求が可能
- データ送受信結果を得ることができる.
  - データポートはデータを送りつけるだけ
- データ送受信以外の処理が可能.
  - アクティビティの制御など
- 実装には，CORBAの知識が必要.

サービスポートの概念

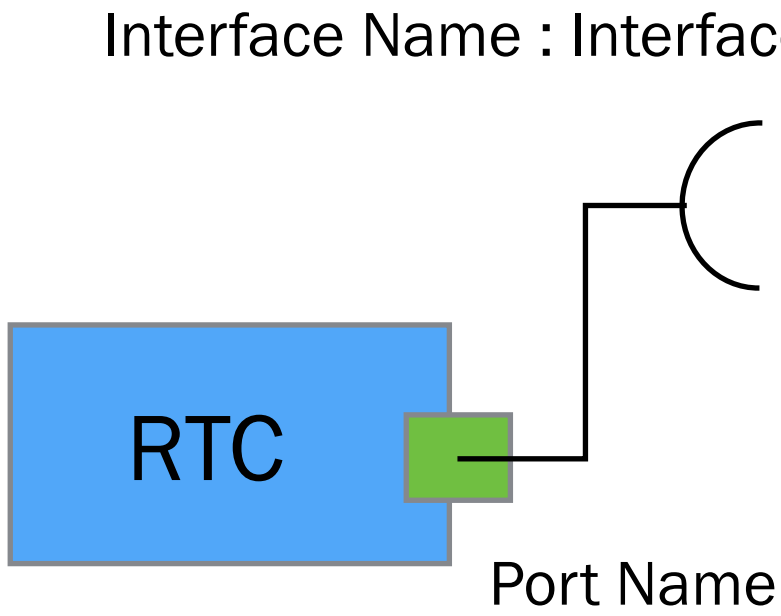
- サービスポートは「インターフェース」を持つ
- インターフェースには極性(Polarity)がある
  - 提供(Provided)
    - リソースを提供する．関数が呼ばれる側．
  - 要求(Required)
    - リソースを利用する．関数を呼ぶ側．
- 一つのサービスポートが複数の提供・要求インターフェースを持つことができる
- インターフェースのタイプが合えば接続でき，利用できる．



サービスポートの作り方

RTC Builderを使うときに必要となる情報

- サービスポートを追加
  - 設定項目
    - サービスポート名
  - インターフェースを追加
    - 設定項目
      - インターフェース名*
      - インスタンス名
      - 極性 (Polarity)
      - インターフェース型



- 注意！
  - インターフェース名が違うと繋げるのが面倒になる！
- インターフェース型を定義するファイルを作成 (IDLファイル)

IDL(Interface Definition Language)

- サービスポートの関数やデータ型を宣言するためのファイル形式（.idlファイル）
  - CORBAやDDSなどの分散処理実現のために使われる
  - オブジェクト指向
- IDLファイルをコンパイルすると，C++やJava，Pythonのプログラムのスケルトンが生成される
  - IDLは中間言語
  - C++やJavaなどの実装言語に変換すると，各言語ごとに若干利用方法が変わるが，関数名などは変わらない
  - C++版について解説

IDLファイル作成の基礎

- moduleで名前空間を定義
  - 後述のインターフェース名は同じ名前になりがちなので名前空間で分ける
- interfaceでインターフェース名を定義
  - 機能をグループ化
- interface内で関数を定義
- 返り値，引数リスト，Javaとほぼ同じ構文
- 引数にin/outを設定
  - in ・・・インターフェース提供側に引数を渡す（値渡し）
  - out ・・・インターフェース提供側が引数にデータを設定する（参照渡し）
  - inout ・・・インターフェース提供側に引数データを渡し，提供側がさらにそれを変更する

```
// 行コメント

// moduleはC++のネームスペースに似た概念
module my_module {


    // interfaceはJavaのインターフェース，
    // C++の仮想クラスに似ている．
    // メンバ関数を定義する
    interface MyInterface {


        // 返り値でデータを受け取るタイプの関数
        long getLongData();

        // 引数でデータを与えるタイプの関数
        void setLongData(in long data);

        // 引数に参照渡しでデータを
        // 受け取ることもできる
        void getDataByArg(out long data);

    };
};
```

IDLファイルの書き方

- structで構造体を定義できる
  - 異なるタイプのデータの集まり
- 引数にも，返り値にも使える

- structでデータ型を定義できれば，データポートの独自データ型作成ができる
  - 参考: http://ysuga.net/?p=213

```
module my_module {

    // 構造体を宣言することでプログラムを構造化できる
    struct MyData {
        long data1;
        long data2;
        double data3;
    };

    interface MyInterface {


        // 返り値でデータを受け取るタイプの関数
        MyData getMyData();

        // 引数でデータを与えるタイプの関数
        void setMyData(in MyData data);

        // 引数に参照渡しでデータを受け取ることもできる
        void getMyDataByArg(out MyData data);

    };
};
```

IDLファイルの書き方

- typedefでデータ型の名称変更
- enumで定数を作る
- sequenceで可変長のデータを作成できる

```
module my_module {

        // typedef を使ってラベルを変更できる.
        typedef long ERROR_CODE;

        // enumを使って数にラベルを付けられる
        enum MY_STATE {
                STATE_RUNNING,
                STATE_HALT,
                STATE_ERROR
        };

        enum RETURN_CODE {
                RETURN_OK,
                RETURN_ERROR
        };

        struct MyData {

                ERROR_CODE errorCode;

                My_STATE state;

                sequence<double> data;
                // sequenceを使うと可変長の配列を使う
        };

        interface MyInterface {

                // 引数でデータを与えるタイプの関数
                // 返り値で処理の成否を返すことができる
                RETURN_CODE setMyData(in MyData data);

                // 引数に参照渡しでデータを受け取ることもできる
                // 返り値で処理の成否を返すことができる
                RETURN_CODE getMyDataByArg(out MyData data);
        };
};
```

サービスポートを用いたRTC作成

IDLファイルを作成する

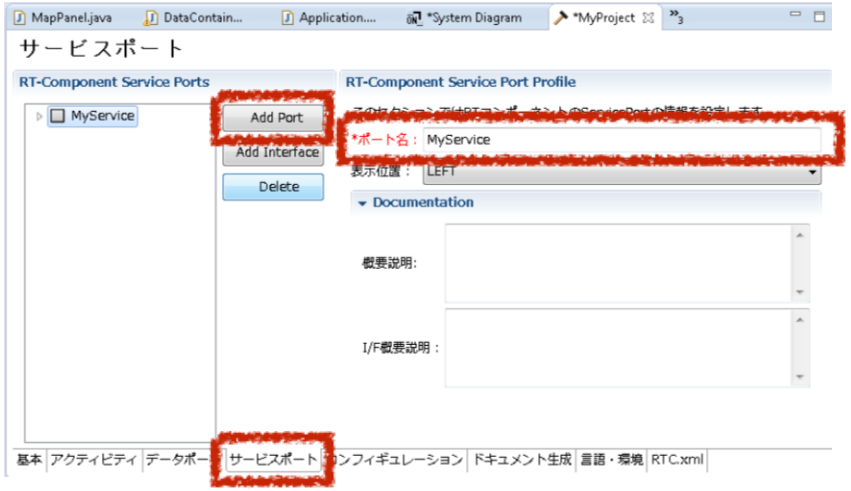
IDLファイル名をtest.idlとする.
C:¥idlというディレクトリを作成し，そこに保存する.

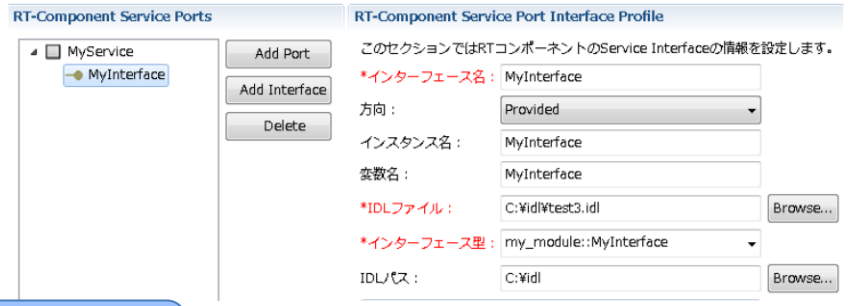
```
module my_module {

	// typedef を使ってラベルを変更できる.
	typedef long ERROR_CODE;

	// enumを使って数にラベルを付けられる
	enum MY_STATE {
		STATE_RUNNING,
		STATE_HALT,
		STATE_ERROR
	};

	enum RETURN_CODE {
		RETURN_OK,
		RETURN_ERROR
	};

	struct MyData {

		ERROR_CODE errorCode;

		My_STATE state;

		sequence<double> data;
		// sequenceを使うと可変長の配列を使う
	};

	interface MyInterface {

		// 引数でデータを与えるタイプの関数
		// 返り値で処理の成否を返すことができる
		RETURN_CODE setMyData(in MyData data);

		// 引数に参照渡しでデータを受け取ることもできる
		// 返り値で処理の成否を返すことができる
		RETURN_CODE getMyDataByArg(out MyData data);
	};
};
```

RTC Builderを用いたRTCひな形作成

第2部を参考に，コンポーネントのひな形を作成する．
基本タブで情報を入力，言語タブでC++を選択し，サービスポートタブで下記の情報を入れる．既存のIDLファイルからRTCを作る場合はここから行う．

- 「サービスポート」タブ選択
- 「Add Port」でポート追加
- 「ポート名」を設定
- 「Add Interface」でインターフェース追加
- 「インターフェース名」「方向」「インスタンス名」「変数名」を設定
- 「IDLファイル」の「Browse」ボタンでIDLを読み込む
- 「インターフェース型」を選択
- 「IDLパス」は「IDLファイルがあったフォルダを選択」





ConsumerとProviderの両方のひな形を作成

ServiceProviderの作成

```
module my_module {

	// typedef を使ってラベルを変更できる.
	typedef long ERROR_CODE;

	// enumを使って数にラベルを付けられる
	enum MY_STATE {
		STATE_RUNNING,
		STATE_HALT,
		STATE_ERROR
	};

	enum RETURN_CODE {
		RETURN_OK,
		RETURN_ERROR
	};

	struct MyData {

		ERROR_CODE errorCode;

		My_STATE state;

		sequence<double> data;
		// sequenceを使うと可変長の配列を使う
	};

	interface MyInterface {

		// 引数でデータを与えるタイプの関数
		// 返り値で処理の成否を返すことができる
		RETURN_CODE setMyData(in MyData data);

		// 引数に参照渡しでデータを受け取ることもできる
		// 返り値で処理の成否を返すことができる
		RETURN_CODE getMyDataByArg(out MyData data);
	};
};
```

<RTC名>_impl.cppの中に書く

```
my_module::RETURN_CODE MyInterfaceSVC_impl::setMyData(const my_module::MyData& data)
{
	my_module::RETURN_CODE result;

	if (data.errorCode == 0) {
		result = my_module::RETURN_OK;
	} else {
		result = my_module::RETURN_ERROR;
	}

	for(int i = 0;i < data.data.length();i++) {
		std::cout << "Data " << i << " is " << data.data[i] << std::endl;
	}

	return result;
}

my_module::RETURN_CODE MyInterfaceSVC_impl::getMyDataByArg(my_module::MyData_out data)
{
	my_module::RETURN_CODE result = my_module::RETURN_OK;
	my_module::MyData_var myData = new my_module::MyData;

	myData->state = my_module::STATE_RUNNING;
	myData->errorCode = 0;
	myData->data.length(4);
	myData->data[0] = 0.1;
	myData->data[1] = 0.3;
	myData->data[2] = 0.5;
	myData->data[3] = 0.7;

	data = myData._retn();

	return result;
}
```

シンプルな関数呼び出しになる
out型引数は_var型を使う

ServiceConsumerの作成

```
module my_module {

	// typedef を使ってラベルを変更できる.
	typedef long ERROR_CODE;

	// enumを使って数にラベルを付けられる
	enum MY_STATE {
		STATE_RUNNING,
		STATE_HALT,
		STATE_ERROR
	};

	enum RETURN_CODE {
		RETURN_OK,
		RETURN_ERROR
	};

	struct MyData {

		ERROR_CODE errorCode;

		My_STATE state;

		sequence<double> data;
		// sequenceを使うと可変長の配列を使う
	};

	interface MyInterface {

		// 引数でデータを与えるタイプの関数
		// 返り値で処理の成否を返すことができる
		RETURN_CODE setMyData(in MyData data);

		// 引数に参照渡しでデータを受け取ることもできる
		// 返り値で処理の成否を返すことができる
		RETURN_CODE getMyDataByArg(out MyData data);
	};
};
```

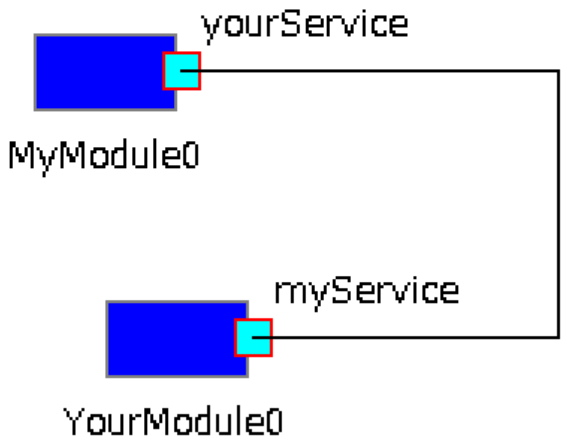
OnExecuteに以下のコードを書き込む

```
my_module::MyData data;
data.errorCode = 0;
data.state = my_module::STATE_HALT;
data.data.length(3);
data.data[0] = 1.2;
data.data[1] = 1.4;
data.data[2] = 1.8;

if(m_variableName->setMyData(data) != my_module::RETURN_OK) {
	std::cout << "[YourModule] setMyData failed." << std::endl;
}


my_module::MyData_var yourData = new my_module::MyData();
if(m_variableName->getMyDataByArg(yourData) != my_module::RETURN_OK) {
	std::cout << "[YourModule] getMyData failed." << std::endl;
}

std::cout << "[YourModule] getMyData succeeded." << std::endl;
std::cout << " errorCode : " << yourData->errorCode << std::endl;
std::cout << " state     : " << yourData->state << std::endl;
std::cout << " data      : [";
for(int i = 0;i < yourData->data.length();i++) {
	std::cout << yourData->data[i] << ", ";
}
std::cout << "]" << std::endl;
```

シンプルな関数呼び出しになる
out型引数は_var型を使う

コンポーネントの動作テスト

- 作成したコンポーネントを起動し，接続して動作を確認する．
  - 右のは一例



自分で作成したインタフェースを用いたRTCの作成方法の整理
・IDLファイルを作成する．
・どこかにそのIDLファイルを保存する．
・RTC Builderを用いて，そのIDLファイルの指定や，IDLファイルのある場所の指定を行う．
- ひな形を出力して，Provider，Consumerのそれぞれに合わせたコーディングを行う．
  - Providerの方では，呼び出されたときの動作を記述
  - Consumerの方では，呼び出す関数を記述

IncludeするIDLファイルは，セットで管理するようにした方が，トラブルが少ない．

まとめ

- ロボットアーム動作RTCの動作検証，実装を通じて，RTC開発の基礎を体験.
- サービスポートの作り方を説明し，サービスポートのテストRTCを体験